(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 456 249 A2**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **91107604.0**

(51) Int. Cl.[5]: **G06F 9/46**

(22) Date of filing: **10.05.91**

(30) Priority: **10.05.90 US 521543**

(43) Date of publication of application:
13.11.91 Bulletin 91/46

(84) Designated Contracting States:
DE FR GB

(71) Applicant: **Hewlett-Packard Company**
Mail Stop 20 B-O, 3000 Hanover Street
Palo Alto, California 94304(US)

(72) Inventor: **Pham, Thong**
10148 Judy Avenue
Cupertino, California 95014(US)
Inventor: **Gulland, Scott**
3681 Irlanda Way
San Jose, California 95125(US)
Inventor: **Amino, Mitch**
373 Bundy Avenue
San Jose, California 95117(US)

(74) Representative: **Baillie, Iain Cameron et al**
c/o Ladas & Parry Isartorplatz 5
W-8000 München 2(DE)

(54) System for integrating application programs in a heterogeneous network enviroment.

(57) A system which integrates applications that run on a plurality of homogenous or heterogeneous computers on a network. System Configuration files (510) in source code are created from a high level definition of the distributed system (LAN) which is to be integrated. The configuration files (510) include data such as the types and formats of data for each process (402) on each node (400) of the system, identification of all applications and machine types, topography and the data manipulations needed for sending messages and files and the like from an application program in a first computer language and of a first data type to an application program in a second computer language and of a second data type. Node-specific data manipulation modules (DMM 528) are formed at each node (400) during startup of the system, and these modules are automatically distributed to nodes (400) on the network having the same architecture. The invention allows applications having different physical data characteristics to communicate by using the data manipulation modules (DMM 528) so formed to manipulate the data at the source program into a common data representation (CDR) having data types common to all of the languages represented by the system and then reconverting the data to the local representation at the destination node.

EP 0 456 249 A2

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention relates to a system for integrating existing application programs in a networked environment, and more particularly, to a system with mechanisms for transforming and manipulating data messages for transfer between different applications on the same computer or on different computers connected via a network or networks and having the same or different computer architectures.

Description of the Prior Art

Since the beginning of the computer age, computers and, in particular, computer software programs have been used in a variety of settings to automate processes which were previously conducted mechanically. This automation has typically led to improved efficiency and increased productivity. However, because of the costs of such automation, automation of large businesses and factories has often been conducted on a piecemeal basis. For example, different portions of an assembly line have been automated at different times and often with different computer equipment as a result of the varying functionalities of the various computer systems available at the time of purchase. As a result, many assembly lines and businesses have developed "islands of automation" in which different functions in the overall process are automated but do not necessarily communicate with one another. In addition, in the office environment LANs have been used to allow new computer equipment to communicate; however, software applications typically may not be integrated because of data incompatibilities.

Such heterogeneous systems pose a significant problem to the further efficiencies of automation since these different "islands of automation" and machines with incompatible data types connected to the same network cannot communicate with one another very easily. As a result, it has been difficult and expensive to control an entire assembly line process for a large manufacturing facility from a central location except on a piecemeal basis unless the entire factory was automated at the same time with homogeneous equipment which can intercommunicate. Thus, for those businesses and factories which have already been automated on a piecemeal basis, they are faced with the choices of eliminating all equipment so that homogeneous equipment may be substituted therefor (with the associated prohibitive costs) or waiting for the existing system to become obsolete so that it can be replaced (again at significant expense).

One solution to the above problem has been to hire software programmers to prepare custom code which allows the different "islands of automation" to communicate with each other. However, such an approach is also quite expensive and is rather inflexible and assumes that the overall system remains static. In other words, when further equipment and application software must be integrated into the overall system, the software programmers must be called back in to rewrite the code for all applications involved and to prepare additional custom code for interface purposes. A more flexible and less expensive solution is needed.

The integration of existing heterogeneous applications is a problem which has yet to be adequately solved. There are numerous major problems in such integration of existing applications because of the differences in hardware and their associated operating systems and because of the differences in the applications themselves. For example, because computers are built on proprietary hardware architectures and operating systems, data from applications running on one system is often not usable on another system. Also, programmers must frequently change application code to create interfaces to different sets of network services because of the diversity of such network services. In addition, different applications use different data types according to their specific needs, and, as a result, programmers must alter a receiving application's code to convert the data from another application into types that the receiving application can use. Moreover, incompatible data structures often result because of the different groupings of data elements by the applications. For example, an element with a common logical definition in two applications may still be stored in two different physical ways (i.e., application A may store it in one two-dimensional array and application B may store it in two one-dimensional arrays). Moreover, applications written in different languages usually cannot communicate with one another since data values are often interpreted differently. For example, C and FORTRAN interpret logical or boolean values differently.

Partial solutions to the above problems have been proposed to provide distributed networks for allowing various applications to share data. In doing so, these applications have relied on transparent data sharing mechanisms such as Sun Microsystems' Network File System (NFS), AT&T's Remote File Sharing (RFS), FTAM (as defined by the MAP/TOP specifications), or Apollo's Domain File System. However, these systems are limited in that they allow data sharing but do not allow true integration of the different

application programs to be accomplished.

Another example of a system for providing interprocess communication between different computer processes connected over a distributed network is the Process Activation and Message Support (PAMS) system from Digital Equipment Corp. This system generally allows processes to communicate with each other regardless of where the processes reside on a common network. Such processes may be located on a single CPU or spread across workstations, clusters, or local or wide area networks (LANs or WANs). The PAMs system manages all connections over the network and provides integration features so that processes on respective workstations, clusters and the like may communicate. In particular, the PAMs message processing system is a network layer which is implemented above other networks to transparently integrate new networks and events into a common message bus. Such a system enables network configuration to be monitored and message flow on the message bus to be monitored from a single point. The result is a common programming interface for all host environments to which the computer system is connected. Thus, all host environments appear the same to the user.

For example, an ULTRIX host environment running ULTRIX-PAMS is directly connected to a VMS host running VAX-PAMS on its networks, and ULTRIX-PAMS uses VAX transport processes to route all messages over the network. Specific rules are then provided for routing messages using ULTRIX-PAMS and VAX transport processes, where the ULTRIX-PAMS functions as a slave transport in that it can only communicate to other PAMS processes via the network to a full function PAMS router. As a result, the PAMS system is limited in that there is no support for "direct" task-to-task communications between ULTRIX processes. In addition, since all traffic must be routed through a VAX-PAMS routing node, a single point of failure exists for the system.

Other systems have been proposed for an information processing environment in which various machines behave as one single integrated information system. However, to date such systems are limited to connecting various subroutines of homogeneous applications running on different machines connected to a common network. For example, the Network Computing System (NCS) of Apollo is a Remote Procedure Call (RPC) software package which allows a process (user application) to make procedure calls to the services exported by a remote server process. However, such RPC systems are typically not fit for the development of a networked transaction management system, for NCS does not provide a message and file handling system, a data manipulation system, a local and remote process control system and the like which allows for the integration of existing applications. Rather, NCS allows for the building of new distributed applications, and does not provide for the integration of existing heterogeneous applications. RPCs instead isolate the user from networking details and machine architectures while allowing the application developer to define structured interfaces to services provided across the existing network.

RPCs can be used at different levels, for the RPC model does not dictate how they should be used. Generally, a developer can select subroutines of a single application and run them on remote machines without changing the application or subroutine code. The simplest use of RPCs is to provide intrinsic access to distributed resources which are directly callable by an application, such as printers, plotters, tape drives for backup tasks, math processors for complex and time-consuming applications, and the like. A more efficient use of RPC at the application level would be to partition the application so that the software modules are co-located with the resources that they use. For example, an application which needs to extract data from a database could be partitioned so that the modules which access the database could reside on the database machine.

A diagram of NCS is shown in FIG. 1. The system 100 therein shown generally consists of three components: an RPC run time environment 132, 134 which handles packaging, transmission and reception of data and error correction between the user and server processes; a Network Interface Definition Compiler (NIDC) 136 which compiles high-level Network Interface Definition Language (NIDL) into a C-language code that runs on both sides of the connection (the user and server computers); and a Location Broker 128 which lets applications determine at run time which remote computers on the network can provide the required services to the user computer. In particular, as shown in FIG. 1, a user application 102 interfaces with a procedure call translator stub 104 which masquerades as the desired subroutine on the remote computer. During operation, the RPC run time system 106 of the user's computer and the RPC run time system 108 of the server system communicate with each other over a standard network to allow the remote procedure call. Stub 110 on the server side, which masquerades as the application for the remote subroutine 112, then connects the remote subroutine 112 across the network to the user's system.

The NCS system functions by allowing a programmer to use a subroutine call to define the number and type of data to be used and returned by the remote subroutine. More particularly, NCS allows the application developer to provide an interface definition 114 with a language called the Network Interface Definition Language (NIDL) which is then passed through NIDL compiler 116 to automatically generate C

source code for both the user and server stubs. In other words, the NIDL compiler 116 generates stub source code 118 and 120 which is then compiled with RPC run time source code 122 by C compilers 124 and 126 and linked with the application 102 and user-side stub 104 to run on the user's machine while the subroutine 112 and its server-side stub 110 are compiled and linked on the server machine. After the application 102 has been written and distributed throughout the network, location broker 128 containing network information 130 may then be used to allow the user to ask whether the required services (RPC) are available on the server system.

Thus, with NCS, the NIDL compiler automatically generates the stubs that create and interpret data passed between an application and remote subroutines. As a result, the remote subroutine call appears as nothing more than a local subroutine call that just happens to execute on a remote host, and no protocol manipulations need to be performed by the application developer. In other words, the NCS system is primarily a remote execution service and does not need to manipulate data for transfer by restructuring a message to allow for conversion from one data type to another. A more detailed description of the NCS system can be found in the article by H. Johnson entitled "Each Piece In Its Place," Unix Review, June 1987, pages 66-75.

The RPC system of the NCS primarily provides a remote execution service which operates synchronously in a client/server relationship in which the client and server have agreed in advance on what the requests and replies will be. Applications must be developed specifically to run on NCS or substantially recoded to run on NCS. Moreover, because a remote procedure cannot tell when it will be invoked again, it always initiates communications at the beginning of its execution and terminates communications at the end. The initiation and termination at every invocation makes it very costly in performance for a remote procedure to set up a connection with its caller. As a result, most RPC systems are connectionless. This is why RPC systems such as NCS must build another protocol on top of the existing protocol to ensure reliability. This overhead causes additional processing to be performed which detracts from performance.

Accordingly, although NCS provides a consistent method for remote execution in a heterogeneous network environment, it is designed primarily to broker distributable services such as printing and plotting across the network, where the user may not care which printer prints the information as long as it gets printed. Another type of service might be providing processing time for applications where a small amount of data in a message can trigger an intensive and time consuming calculation effort to achieve an answer that can itself be turned into a message. However, the NCS system cannot provide a truly integrated system for incompatible node type formats and data processing languages.

None of the known prior art systems address the substantial problems of integrating existing heterogeneous application in a heterogeneous and/or homogeneous network environment. Accordingly, there is a long-felt need in the art for an integration system which provides for flexible data transfer and transformation and manipulation of data among existing applications programmed in a networked environment of heterogeneous and/or homogeneous computers in a manner that is transparent to the user. The present invention has been designed to meet these needs.

SUMMARY OF THE INVENTION

The inventors of the subject matter disclosed and claimed herein have satisfied the above-mentioned long-felt needs in the art by developing a software tool which enables a system integrator or end-user flexibly and efficiently to produce run time software for integration of existing applications in a networked environment of heterogeneous computers. In order to achieve this goal, the present invention provides functionality equivalent to that of a combination of a message and file handling system, a data manipulation system, and a local and remote program control system. From the system integrator's viewpoint, the present invention provides a message handling system which allows data types and data formats to be different at each end of the messaging system, while any changes in data elements, data types or data formats of the messages will only require a reconfiguration of the system before start-up. Since reconfiguration is an administrative level activity, the user will not be required to change his or her source code in the communicating applications.

Accordingly, the present invention is specialized for easy modification of data types and data formats passed so as to allow transparent communication between data processes of different formats on machines of different types. As a result, the application programs which are communicating need not be written in the same language or be downloaded onto the same computer type. The present invention further allows users to link existing applications with minimal or no changes to the code of the applications, thereby reducing the amount of custom code that needs to be written, maintained and supported for integrating existing systems.

The present invention addresses the major integration problems noted in the background portion of this

specification by providing for local and remote inter-application data transfer whereby existing applications may be linked with minimal or no modifications to the applications. Synchronous and asynchronous memory-based message transfers and file transfers between applications are also supported. In addition, language, data format and data type differences are resolved utilizing data manipulation features such as rearranging, adding or deleting fields and converting between data types in accordance with differences in hardware, language alignment, and data size. This is accomplished in a preferred embodiment by using a Common Data Representation (CDR) for the messages to be transferred between heterogenous nodes.

The data manipulator (DMM) of the invention provides automatic manipulation of data during run time so that two communicating processes can use each other's data without having to change their own data models. The data manipulator of the invention takes care of hardware discrepancies, application dependencies and computer language semantic differences. It can convert one data type to another, restructure a message format and assign values to data items.

Typically, conversion routines are only good for the two machine architectures and/or two languages involved. With the addition of any new language or machine architecture to this networked system, a new set of routines must be created on all previous machine architectures in the network to support the transfer of the data to applications on the new machine or to applications written in the new language. The present invention has been designed to minimize the alteration or addition of routines that were written on older machine architectures in the network when new machines or languages are added to the system. Also, by making the data manipulation module node-specific, it is also possible in accordance with the invention to cut down on the number of sets of routines a particular machine might need to send/receive data to/from other machines.

BRIEF DESCRIPTION OF THE DRAWINGS

The objects and advantages of the invention will become more apparent and more readily appreciated from the following detailed description of presently preferred exemplary embodiments of the invention taken in conjunction with the accompanying drawings of which:

FIG. 1 schematically illustrates a prior art Network Computing System (NCS) which allows applications running in a distributed environment to share computations as well as data;

FIG. 2 schematically illustrates the basic components of the integration system in accordance with the invention;

FIG. 3 schematically illustrates how the invention can be used to connect an application to others on the same system or to applications that reside on one or more remote systems.

FIG. 4 schematically illustrates the configuration of the run time components of the integration system in accordance with the invention;

FIG. 5 schematically illustrates the creation of a Data Manipulation Module (DMM) of the invention through start-up by the start-up node;

FIG. 6 schematically illustrates the distribution of the configuration information from the compilation node to each of the respective nodes of the same computer architecture as the compilation node;

FIG. 7 is a flowchart illustrating the procedure for start-up of the integrated system which uses this invention;

FIG. 8 schematically illustrates an example of a heterogeneous networking system in accordance with the invention; and

FIG. 9 illustrates sample configuration files for the sample configuration shown in FIG. 8.

DETAILED DESCRIPTION OF THE PRESENTLY PREFERRED EMBODIMENT

A system with the above-mentioned beneficial features in accordance with a presently preferred exemplary embodiment of the invention will be described below with reference to FIGS. 2-9. It will be appreciated by those of ordinary skill in the art that the description given herein with respect to those figures is for exemplary purposes only and is not intended in any way to limit the scope of the invention. All questions regarding the scope of the invention may be resolved by referring to the appended claims.

As noted above, present day manufacturing computer systems have often been built from the bottom up, thereby resulting in the creation of isolated "islands of automation." For example, the areas of design and engineering, manufacturing resource planning, and manufacturing have typically been independently automated. As a result, these islands of automation consist of heterogeneous computer systems, operating systems, and data base systems. However, manufacturers are now looking for higher levels of efficiency and productivity than is available in such prior art systems. Computer Integrated Manufacturing (CIM), the

integration of these islands of automation, is a means of achieving such higher levels of efficiency and productivity. The preferred embodiment of the present invention is designed for facilitating the development of CIM solutions. As will be clear from the following, this is accomplished by integrating existing application programs in a networked environment of heterogeneous computers by providing flexible data transfer, transformation and manipulation mechanisms.

Generally, the present invention produces run time code that comprises several active programs (or software modules) which are used at run time to provide communication between applications on the same or different nodes, where a node is any computer in the network's domain. The applications can be on different computer systems or on the same computer and may be in the same or different computer languages. These run time programs handle the data transfer from the source application program to the destination application program by transforming the data from the source program into an architectural and language independent format using a CDR. For example, the field and record data in the source machine's format is converted to the destination machine's format by first converting to a CDR using locally stored routines which convert the internal representation of data on the source machine to the common data representation format, and after the transfer of the CDR data, the CDR data is converted to the internal representation for data on the destination machine using locally stored routines. In particular, predefined links of sources and destinations are called by software modules of the invention to make certain that the information is correctly routed from the source to the destination machine. This latter information is called the configuration data and corresponds to information provided by the system's integrator concerning the nodes, processes and possible data manipulations to be performed on the network.

The system of the invention generally consists of run time and non-run time components. The run time components of the invention will be described in more detail with respect to FIGS. 2-4, while the non-run time components will be described in more detail below with respect to FIGS. 5 and 6.

As shown in FIG. 2, the run time components 208 include a data manipulator/translator, a data transporter, configuration tables and a system manager. As will be described in more detail below, the data manipulator/translator functions to transform data to and from a CDR format acceptable to a given process of a specified host computer, while the data transporter functions to pass data in the common data representation to/from the network system for sending to a destination application. However, as will be noted below, the data transporter also functions to send unmanipulated data to the destination application when the source and destination applications have the same type and format and hence the message data does not need to be manipulated.

Thus, as shown in FIG. 2, in the system of the invention a plurality of user application programs 202 are connected via application adaptors 204 using access routines 206 so as to communicate with the run time components 208 of the invention. The application adaptors 204 comprise user-written programs which act as software bridges between the user's application and the system of the invention. In particular, these user-written application adaptor programs call access routines from an access routine library 206 in order to send messages, copy files, control processes, and the like on the network of the invention. These access routines 206 are thus used to provide the user with the necessary commands for preparing the application adaptor programs. Sample access routines are provided below as an Appendix A hereto, and sample adaptors are attached as an Appendix B hereto.

System manager 210 enables a user to administer operation of the system during run time operation. For example, the system manager 210 allows the user to access the software of the invention to validate configuration, start-up and shut-down the system and perform other system management functions. The command processor 212, on the other hand, corresponds to a module accessed through the system manager 210 so as to allow a user to interactively test the messages and links to any process or node within the domain of the invention. (As used herein "domain" means the collection of computers that are configured to work together in accordance with the invention.) In addition, configuration files 214 contain information about the nodes, the structure of the data produced and consumed by the processes, the manipulations that must be performed on data produced by an application so that it is acceptable by different processes, and the links binding manipulations to specific languages. Such information is provided by the systems integrator before run time and will be discussed in more detail below. The run time components are loaded into each node at start-up time using the system manager 210, and these components manipulate messages (as necessary) and transport them using the data transporter. The manipulations are performed in accordance with data in configuration files 214 held in memory at each node. Finally, the data transporter communicates with the network services 216 to pass the message to the destination node.

FIG. 3 shows a system in accordance with the invention where two nodes on a LAN have run time integration systems A and B in accordance with the invention for integrating applications 1-4. As shown,

each node is loaded with the run time system of the invention (including all elements shown in FIG. 2) for allowing the processes 1-4 to communicate even if they have different data types and language formats. In FIG. 3, there are various possibilities for communication between the processes shown. For example, Application 1 may communicate with Application 2 through the integration System A. Application 1 may generate data meant for Application 2 in the shared memory of the host computer or in the form of files resident on the host disk. This data is picked up by Application Adaptor 1 and passed to Integration System A, which then gives it to Application Adaptor 2 for passing on to Application 2. Integration System A optionally can be instructed to manipulate the data before sending it to Application 2. The data manipulation can consist of literal assignments to data, moving data from one structure to another, and translating data from one structure to another.

On the other hand, the data may be picked up by Application Adaptor 1 and given to Integration System A. This data is then sent across the LAN to Integration System B. Application Adaptor 3 is given the data by Integration System B which then passes it Application 3 for its use. The user can decide whether any necessary data manipulations are done by Integration System A or Integration System B.

FIG. 4 shows an implementation of the present invention on a given node 400 of a heterogeneous network system during run time. As shown, the integration system of the invention integrates all of the user's application programs (processes) 402 which run on the node 400. Applications running on other nodes (not shown) are similarly integrated. As noted above with respect to FIG. 2, these application programs 402 are respectively given access to the integration system of the invention via access routines 404, which communicate with the integration system of the invention via a library of access routines such as those in Appendix A.

The request manager (RM) 406 maintains a single queue for all user's application program requests so that one entry per defined process on node 400 may be read at a given time. The request manager 406 reads the request from the application program 402 and performs the requested action. Requests can be of the following types: reading a message, sending a message, sending a file, starting another process, stopping another process, clearing the RM message queue area, recording an error on spooling device 410, setting the options for data transfer, deleting a particular message and the like, as apparent from the sample access routines in Appendix A. For example, a user's application program 402 may call an access routine to send a message to an application program on another node to indicate that a value monitored by the application program 402 has changed state. The request manager 406 will then check its configuration tables 408 and the request from the application adaptor to see if a data manipulation was requested and is necessary for the sending of the message to the remote node. As will be described below with respect to FIG. 5, the configuration tables 408 will contain information about all nodes, processes and data manipulations possible for all message transfers within the network. Accordingly, by determining the source process and the destination process of the message request as well as the respective nodes and whether the message requested specified a link logical name, the request manager 406 can determine whether the data needs to be manipulated for the transfer. The request manager 406 also maintains a message queue (memory area) 409 which contains linked lists of data messages for incoming data from other processes. In addition, request manager 406 may log user messages to disc 410, if requested, by the sending process. Information such as error messages and the like also may be stored for use during system diagnostics.

When request manager 406 determines that an indicated message needs to be manipulated (i.e., the source message needs to be converted to a different format and/or different language), the request is passed to data manipulation module (DMM) 412 where the message is converted to a common data representation (CDR) as will be described below before being sent to the destination node. If the message from the application program 402 calls for a transfer of a data file, the indicated file is transferred by File Transfer Module (FTM) 414 to the destination node. A file message containing file information is sent to ONM 416 after a successful file transfer. If the request manager 406 determines that the application program is to send a message to a destination process without manipulation, no manipulation is performed. The message data is thus sent directly to Outgoing Network Manager (ONM) 416. ONM 416 receives the unmanipulated message from request manager 406, a message from the File Transfer Module 414 indicating that a file has been transferred or the message converted to its common data representation by data manipulation module 412 and outputs the message onto the LAN by making a network service call to send the message to the destination node. Messages that are specified as "critical" by the sending process and which cannot be successfully sent over the network to the destination node are stored (spooled) by ONM 416 to disc file 420. Failure to send a message can occur for several reasons including a node or network failure or a system shutdown. ONM re-attempts to send a message previously spooled to disc 420 when the original cause of the failure to send the message has been corrected.

If the sending process specifies that a message is to be sent "with-wait", then ONM 416 sends status

information to request manager 406's input queue when the message could not be sent to the destination node. However, if a message does not have to be manipulated and the destination process resides on the same node, then the message is not sent via ONM 416. Instead, the message is placed directly into the destination process' message area queue, preventing unnecessary data transmission. If the sending process specifies that a message is to be sent with "guaranteed delivery", then the request manager 406 will save those received messages in disc file 410 and will delete such messages only if there is a deletion request from an application program 402. A read request is typically done by a receiving process before a deletion request is done. If a message is not to be sent with guaranteed delivery, then request manager 406 saves the message in its memory area 409 and the message is available to the receiving process via a read message request. When the receiving process does a read request, it can specify whether the message is to be deleted or not.

Incoming network manager (INM) 418 receives incoming messages from the network and determines whether these messages need to be converted to local format from the common data representation format before being sent to the destination application program 402. If such a translation is necessary, as when the source node was of a different data type or the source programming language was different, the message is converted from the common data representation to the local data representation using a library of conversion (unmarshalling) routines before the message is sent to the input queue of request manager 406. Incoming messages which do not require such manipulation are sent directly to the input queue of the request manager 406.

Finally, as noted above with respect to FIG. 2, a system manager 422 is also preferably provided to allow the user to administer the system management functions of the integration system of the invention during run time. Preferably, a particular node is selected for this purpose.

The manipulations must take place when an application A having a first language A and data fields of a first type is to communicate during run time with an application B having a second language B and data fields of a second type. In accordance with the manipulation technique of the invention, a Data Definition Language (DDL) is used to define the data fields of the application program A and the application program B. The data fields of the DDL are then manipulated (rearranged) or the data types are converted by a Data Manipulation Language (DML) to that of a Common Data Representation (CDR) having a universal encoding scheme such as Abstract Syntax Notation One's (ASN.1) Basic Encoding Rules (BER). The manipulated data is then converted at the destination node into language B with associated data fields for application B. DDL and DML will now be described in more detail.

The aforementioned Data Definition Language (DDL) is a high level language used in accordance with the invention to define the logical structure and types of data used by all application programs on the network. More particularly, DDL is used to define data types with declarations in the Data Definition configuration file (Ddef) as will be described in more detail below with respect to FIG. 5. Traditional languages such as C, FORTRAN, and PASCAL are directly tied to the machine architecture in which they are implemented. This is because while different languages may have identical logical data types, those data types may be physically represented differently. For example, both C and FORTRAN have multi-dimensional array types; however, to store this data type, C uses row-major ordering while FORTRAN uses column-major ordering. DDL provides a full set of predefined data types that lets the system integrator describe the logical layout of the data used in application programs. In addition to the basic data types in most languages, DDL may include an additional data type called "opaque" in accordance with the invention, where "opaque" is a custom data type used to describe "type-less" data.

Data in DDL is defined in a manner roughly equivalent to the type definition in C or the type declaration in Pascal. DDL declarations in accordance with the invention are similar to the data declarations sections of standard languages such as C and PASCAL and hence are believed to be well within the level of skill of one of ordinary skill in the art. For example, the DDL of the invention preferably includes information about the data types of all the procedures, and unlike the prior art, the DDL of the present invention allows for conversion from one data type to another in accordance with the DML described in more detail below. Moreover, unlike traditional languages, a DDL definition in accordance with the invention does not bind any particular language or machine representation to the data definition. Rather, data definitions and machines are linked through a linkage definition as will be described below. Thus, the DDL in accordance with the invention is primarily concerned with the logical layout of data, and accordingly, a suitable DDL which supports the languages on the network to be integrated is believed to be well within the skill of one of ordinary skill in the art. By way of example, a partial description of a possible DDL supporting C, FORTRAN and Pascal on a network is described in Appendix C. Appendix C aiso includes BNF syntax diagrams for DDL declarations in such a case.

Data Manipulation Language (DML) as used herein is a high-level language that is used to manipulate

data. After the data types are defined in DDL (in Ddef), the manipulations that must be performed on that data are defined in the Data manipulation definition configuration file (Dman) using DML as will be described below with respect to FIG. 5. The data manipulation declarations can be either assignment statements or move statements. Assignment statements set default values for destination structures, while move statements move the source data structure to the destination data structure, correcting for differences in languages and machine architectures, and providing type conversions where necessary. These type conversions may be performed on boolean, integer, floating point and ASCII fields.

As an example of a DML in accordance with the invention, a DML is described in Appendix D for a network have a DDL supporting C, FORTRAN and Pascal. Appendix D also includes BNF Syntax diagrams for DML definitions in such a case. One skilled in the art will appreciate that the DML described in Appendix D is for a specific embodiment and may be readily modified by one skilled in the art in accordance with the general principles set forth herein.

SETUP PROCEDURES

FIGS. 5-7 illustrate the procedure for starting the system and setting up the DMM run time module described above with respect to FIG. 4. However, before describing these figures, the basic steps in the task of integration analysis that a system integrator or end user would follow in accordance with the system of the invention will be described. During the process, the system integrator completes a high level design of the integration system of the invention by determining the number and general functions of the processes needed to fulfill defined functional requirements. This information is required in both configuring the system and developing the application adaptors for it.

Basically, configuring the system of the invention consists of creating with any text editor a set of configuration files that describes the data to be transported and the operating environment. The configuration files provide information about the data such as the way it is structured, how to access it, how to manipulate it, and how to display it. They describe the operating environment by defining the computers in the network, their links, and the types of processes. The configuration fields are formed by the system integrator or end user in accordance with the following steps.

1. The Functional Requirements Analysis.

Functional requirements analysis means clearly defining the environment in which integration by the invention is to be performed so that the systems integrator can do a high-level design of the system to be integrated. In other words, the user determines what is to be accomplished by integration so that the requirements of such integration can be specified in a programmatic interface development phase. For example, the user may want several applications that manage part of a manufacturing system, such as tracking and scheduling work orders, to be able to communicate and exchange data. The user will need to know the resources, namely, the applications that perform particular functions or processes, the machine nodes on which those applications reside (i.e., logical node name, machine type, physical node name, and so forth), and network characteristics. Such network information is stored in a network definition (Netw) configuration file.

2. Data Flow and Application Analysis

The user must also analyze the types of data and the movement of the data between the applications that will be integrated. The user also will need to identify any synchronization needed to control the flow of data or to insure that the data remains consistent. Whether the data flows between different applications on the same system or between applications on different systems, the user also must know the specifics of the data being moved between applications. For example, the user must know the structure of messages and source applications sent and what format the destination application will accept. The user must also know the programming language used to produce the source message and what language will be used to read the messages at the destination. If manipulations are necessary to transform the message data into a format acceptable by the receiving application, the user will form a Data Definition file (Ddef) written in DDL and a Data Manipulation file (Dman) written in DML. The Ddef file defines the format of the messages passed between applications, specifically, the integers, strings, arrays, structures, records and bytes that make up the output of one application and the input for another application. The Dman file, on the other hand, contains the definitions of each data manipulation as described above. Each data manipulation defines the operations that must be performed on a source data structure so that it has the format and typing that the

destination application can accept.

3. Designing The Processes

The user designs the integrated system of the invention by determining the tasks or processes required by the flow of data. The user also determines how these tasks should be grouped in application adaptors and stores this information in the Process Definition configuration file (Proc). Information in the Proc file lists the processes that send and receive data, default runstrings and miscellaneous execution information. The process logical name is used by the access routines (Appendix A) to reference a particular process definition entry.

4. Determining Data Format

The user also determines whether the data exchanged is in file or message format. If it is in file format, the user determines what file copy options to use. This information should be stored in a File Definition file (Fdef). If it is in message format, it should determine what manipulations and the like are needed.

5. Setting Links

Finally, the user defines links to associate data definitions and data manipulations with the list of nodes where the source data may be produced and the source and destination languages of the data transferred. This information is stored in a Link Definition configuration file (Link). The Link file describes the physical characteristics of data at the source and the destination of a message and bind specific data or manipulation definitions to particular links. The Link definition binds data definitions and manipulations to source locations and architecture types, source and destination languages and file definitions.

Thus, configuration in accordance with the invention means describing the applications, nodes in the network, processes, files, links and manipulations to the integration device of the invention. This is the information gathering process. In a preferred embodiment of the invention, six configuration files 502 (FIG. 5) are created by a text editor to model the system to be integrated in accordance with the invention.

Once these configuration files are created and validated, the system of the invention is ready for start-up. In particular, the run time system of the invention as shown in FIG. 4 must be established on each node of the network so that the processes on each of the nodes within the network may communicate with each other as specified in the configuration files. In other words, the DMM modules of the invention as shown in FIG. 4 must be made node-specific and installed on all the nodes in the integration domain and the configuration tables 408 loaded into memory. However, before this is done, the integrator designates an administration node, for providing functions such as start-up, shut-down and security, and preferably, a given number of alternate administration nodes are also established in case of failure of the primary administration node. The system manager 422 is then used to configure system security and to perform startup as will be described below. The user then writes and compiles the application adaptors for the various processes. However, since the application adaptors require configuration information, configuration and application adaptor development can be done in conjunction or in any order. On the other hand, since the user may use the command processor to simulate other adaptors for testing purposes, it is preferable that the user perform adaptor development as a final step. The user is now ready to start running the system of the invention to generate the DMM modules as shown in FIG. 5.

As shown in FIG. 5, the configuration files 502 are loaded onto the administration node. A validation program 504 reads in the network definitions (Netw), the process definitions (Proc), the link definitions (Link) and file definitions (Fdef) from the configuration files 502. The configuration files 502 must reside in a directory identified to validation program 504. Validation program 504 checks the integrity of the configuration files and converts them into the binary data files 510 required at start-up. The validation program then invokes the DDL compiler 506 to read the data definition (Ddef) and produce a DDL symbol table in memory. The DML compiler 508 is then invoked by the validation program 504 to read the data manipulation definitions in the Dman file and to generate C source files 512 which contain the C language instructions to perform the data manipulation and conversion of the source data to a common data representation. The DML compiler 508 then uses data definition information for the DML compiler 508, link definitions from validation process 504 and manipulation definitions from Dman 502 to generate the look-up table C source files 530, the manipulation C source files 512, and the C source index files 514. The C source index files contain the list of all C sources generated for the various nodes in the domain. A different source file index is generated for each machine type.

The validated configuration files 510, the manipulation C source files 512, and C source index files 514 and look-up table C source files 530 thus represent the configuration information in a manner that can be loaded into memory but is instead used to build the DMM 412. These files are used by the administration node during start-up process 516 to create Configuration Tables which are distributed to all nodes in the integration domain. The C source index files 514, lookup table C source files 530, and manipulation C source files 512 are used in forming node-specific data manipulation modules.

In particular, start-up is initiated on the administration node by invoking the DDLOAD program 518 to load the Configuration Tables into memory. The Configuration Tables include (1) a list of nodes in the integration domain and (2) a list of compilation nodes, where a compilation node is defined as a node on which the integration system compiles the look-up table C source files 530 and the manipulation C source files 512 for all nodes belonging to that computer architecture family in order to create the node-specific DMM. There must be at least one node defined as the compilation node for each type of computer architecture in the integration domain. In other words, a compilation node is provided for each type of computer architecture within a heterogeneous computer network environment. Of course, for a homogeneous computer network environment, only one compilation node is required.

Once the Configuration Tables are loaded on the administration node during start-up, start-up process 516 marks the nodes to be started and asks the system manager to schedule the DMM builder 520 on each compilation node. As noted above, there must be a compilation node for each machine type in the system. If the compilation node for a particular machine type is not found, an error message is issued and start-up is halted on all nodes of that machine type.

At each compilation node, the following start-up activities are conducted by DMM builder 520. The DMM builder 520 pulls (copies) the C source index file for its particular machine type along with the relevant C sources (look-up and manipulation) and the list of nodes within the network that need a new DMM onto the compilation node and compiles the C source files into object modules. The result is a DMM module 528 on the compilation node which corresponds to the DMM 412 used on that node during run time as shown in FIG. 4. The C compiler 524 compiles the manipulation C source files 512 into a customized library of object files 526 which is specific to that machine or architecture type. The Compiler 524 compiles the look-up table C source files 530 into node specific object files.

The compilation node then distributes the node-specific DMM module 528 to each node of its machine type on the network that needs it. FIG. 6 shows how the compilation node distributes the DMMs to all such nodes. As shown, the validated configuration files 510, the look-up C source files 530, the manipulation C source files 512 and the C source index files 514 are pulled by (copied) the compilation node for forming an architecture specific DMM module 528 as just described. This DMM module 528 is then distributed to all other nodes (nodes 3-6) of that architectural type on the network which need a DMM, and these nodes, in turn, notify the compilation node that they have received the DMM. The compilation node passes this status to the start-up process.

In summary, the process performs the following activities on each node, beginning with the start-up node: copies the validated files to the node; loads the Configuration Tables into memory; prepares the run time programs for start-up; starts/stops the run time programs according to the user's decision; copies a set of configuration files to all alternate administration nodes; and copies the Manipulation C source files 512, the lookup files 530 and the C source index files 514 to all alternate administration nodes.

FIG. 7 is a summary of the procedure for setting up and starting the integration system of the invention. As shown, the user first configures and validates the integration at step 702 and then determines at step 704 whether application adaptor programs need to be created or modified in order to interface the application program to the integration system. If so, the application adaptor programs are created or modified at step 706 and then compiled at step 708 for linkage into the system at start-up (step 710). The user may then test the configuration at step 712 using the command processor, and if the configuration has no problems, all nodes may be started up. On the other hand, if there is a problem with start-up, at step 714 it is determined whether the configuration needs to be changed. If not, the application adaptor programs are modified at step 716, and the modified adaptor programs are then compiled at step 708 and once again linked into the system for start-up at step 710. However, if the configuration does need to be changed after testing, control branches from step 714 to step 718 where the system is shut down to allow for reconfiguration and validation of the Configuration Tables 408. Thus, in the presently preferred embodiment, the system is shut down for reconfiguration since the configuration data is linked in during start-up of the nodes. However, as will be apparent to one of ordinary skill in the art, the configuration may be changed and validated during run time without shutting down the system, but it cannot be switched into the running system unless the current system is shut down. Once the integrated system is running, control may be turned over to the system administrator for daily administrative tasks.

DMM RUN TIME OPERATION

As described with respect to FIG. 5, the data manipulator of the invention preferably comprises a DDL compiler which processes data declarations and generates symbol tables for use by other software modules of the data manipulator. A DML compiler is also provided for processing data manipulation statements defined by the users and for generating C source code modules that will be compiled with data encoding functions (such as ASN.1 Basic Encoding Rules) in a Common Data Representation Library and with data conversion and manipulation functions in a Data Manipulation Library. The object code modules created after the compilation will be executed at run time by a module called the data manipulation shell, which determines what type of manipulations are required for a specific message and invokes appropriate manipulation modules generated by the DML compiler to transform data into a common data representation notation for transmission to a computer with a different architecture. Finally, the data manipulator of a preferred embodiment of the invention will include an "unmarshaller" module which is invoked by the Incoming Network Manager 318 to decode a message in common data representation format to the local host data and language representation.

During operation of the data manipulation module, the data manipulation shell reads a message from its in-coming mailbox (from request manager 306) and calls appropriate data manipulations (such as those in pseudocode form in Appendix E) to transform the data as requested by the user's application program (i.e., to match the destination format). After the message is manipulated, the data manipulation shell sends it to the Outgoing Network Manager 316 for delivery. At the receiving node, the Incoming Network Manager 318 gets the message from the network and determines whether an unmarshalling task (such as those in pseudocode from in Appendix F) is necessary. If it is, the incoming network manager 318 invokes the unmarshaller module. From the message content, the unmarshaller module determines which functions are needed and invokes these functions to perform the unmarshalling.

As noted above, the objective of the data manipulation module is to enable the user to transfer data between two applications residing on different machine architectures and written in different languages. The problem which arises when this is attempted is that the sending and receiving machine architecture/language may emit or expect the data to be in a particular representation or a particular data size or alignment. For example, string representations, boolean representations, row/column major addressing, data size and the like may be language specific, while byte order, two's complement notation, floating point (real) representation, character set, data size and the like may be machine-specific. Such differences in architectures and languages imply that some sort of data manipulation must be performed on data emitted to make it acceptable to a receiving application. Conversion routines are needed to support this data manipulation, and such conversion routines are included within the data manipulation module of the invention as described above with respect to FIGS. 5 and 6.

The present invention uses a Common Data Representation (CDR) in which the conversion routines needed are written to convert data back and forth from a local machine and language format to a machine- and language- independent data format. Thus, a sender of data will convert the data into CDR format and send the data. A receiving system will invoke routines to convert the CDR-formatted data into the receiving system's local data format. Each machine architecture thereby only has to know about how to convert from its local machine format to CDR. Moreover, any new machine architecture added to the network only has to know the same for its data converting.

In accordance with the invention, one skilled in the art may design his or her own common data representation which will handle all the data types which are to be supported on the network. However, a presently preferred embodiment of the present invention uses an already existing, standard CDR encoding rules entitled OSI ASN.1 Basic Encoding Rules (BER). In particular, a preferred embodiment of the invention uses the byte-level "transfer syntax" format specified in ASN.1 for CDR encoding the data described by the data definition and data manipulation languages previously described.

Thus, in accordance with a preferred embodiment of the invention, the data elements transferred between machines are in ASN.1 BER encoding. The problem here is that when an ASN.1-encoded data element arrives on a target system, that data element could be unmarshalled into any one of a number of language- specific data types. Thus, a "type attribute ID" must accompany the ASN.1 encoded user data element in the message to identify the language-specific data type that the ASN.1-encoded user data element will be unmarshalled into on the target system. These type attribute IDs are also in ASN.1 encoding. A mapping is needed between DDL types and ASN.1 data types. The following is a table of presently preferred DDL data types versus ASN.1 encoded data types:

DDL Types	Type Attribute ID	ASN.1 Types	Type Attributes
bit_field	BIT_FIELD (0)	bitstring	
byte (8-bit)	BYTE (1)	integer	
short_int	SHORT_INT (2)		
integer	INTEGER (3)		
long_int	LONG_INT (4)		
short_boolean	SHORT_BOOLEAN (5)	boolean	
boolean	BOOLEAN (6)		
long_boolean	LONG_BOOLEAN (7)		
char (8-bit)	CHAR (8)	string	
opaque (8-bit-aligned)	OPAQUE (9)	octet string	
real	REAL (10)	real	
long_real	LONG_REAL (11)		
string (8-bit chars)	STRING (12)	string	max length
unsigned bit_field	U_BIT_FIELD (13)	bitstring	
unsigned byte (8-bit)	U_BYTE (14)	integer	
unsigned short_int	U_SHORT_INT (15)		
unsigned integer	U_INTEGER (16)		
unsigned long_int	U_LONG_INT (17)		
multiple-dimen array*	ARRAY (18)	sequence-of	
record	RECORD (19)	sequence	restrictive type
packed multiple-dimen array*	PACKED_ARRAY (20)	sequence-of	
packed record	PACKED_RECORD (21)	sequence	restrictive type

TABLE 1

***The unmarshalling module assumes that the code doing the marshalling will place the elements of the multiple-dimensioned array into the row-/column-major order for the target language. In other words, the unmarshaller assumes that the elements of the ASN.1 array are in the proper row-/column-major order for the target language. It does NOT do reordering of array elements.**

The parenthesized ( ) numbers after the type attribute ID name indicate the numerical value of the type attribute ID that will be used in the code.

As shown in Table 1, the type attribute ID is a name identifying the target DDL type of the ASN.1 type. The number after the name is used in the source code to represent that ID. Besides the type attribute IDs, the type attributes also include extra information that is transferred with the ASN.1 data type to allow for the correct unmarshalling by the receiving node into the target DDL type. ASN.1 data type format does not have a place for such information. For example, the ASN.1 string type data element has a field to indicate the current size of the string value, but no field to indicate what the maximum size of the string can be. This maximum size information is type attribute information. Because both the type attribute IDs and type attributes are extra information needed in addition to the actual ASN.1 encoded user data, they have both been conceptually lumped together under the title of "type attributes" herein.

"Marshalling" of data involves converting data in a local format into a machine and language independent CDR format as described above. Such marshalled data can then be sent to an "unmarshalling" module resident on this or another system in the network. The unmarshalling module will then convert that data into the local machine- and language- dependent format. In accordance with the invention, a marshalled message is broken down into two main parts: a static format message header and a dynamic format application data part. The division of these parts is based on whether the part contains predefined, static format data, or whether it contains application-bound data that can have any type of format. The

message header contains administrative information needed to transfer the message, such as source/destination node names, operation flags, target process logical names and the like. It also contains the information necessary to carry out the specific functions such as send a file, start/stop a process, return status to the machine application server, and the like. The dynamic format part, on the other hand, varies depending on the application data being transferred.

The dynamic part of the message contains significant subfields. These are shown as follows:

Static Message Part
Data Buffer Header/User Data/Type Attributes

The data buffer header contains information needed to initialize the unmarshalling process, while the user data is the ASN.1 encoded "pure" data that is expected by the receiving application. Type attributes, on the other hand, are extra, non-ASN.1 information needed to help unmarshall the user data. For example, this type attribute data will be node specific information which allows the user data to be converted to the local data format type. Preferably, the user data is kept in one buffer and the type attribute data is contained in another data buffer during marshalling to facilitate appending (copying) of the type attribute data for a given primitive to the associated user data. The data buffer header, on the other hand, may contain information such as at what offset do the type attributes start and what is their total length. Detailed examples of the implementation of marshalling with this type of buffer format can be found in Appendix E. The user data will be marshalled into ASN.1 data elements according to the DDL-type/ASN.1-type table given previously and the ASN.1 Basic Encoding Rules (BER).

The data manipulation module marshalls the data in accordance with the invention by building a library called a Marshalling Routines Library (MRL). Each MRL will be customized for the particular machine architecture. Thus, there will be one MRL version per machine architecture type supported. In a preferred embodiment, the MRL assumes that it will be called by C language code compiled by a C compiler supported by the integration system for that particular computer architecture. Also, the MRL is not designed to be a run time shared library, and instead, the library code will be linked into the DMM's code at link time as described above.

Code emitted by the DML compiler 508 will do the following marshalling steps:

1. Start the marshalling process by marshalling the header;
2. Marshalling the source data elements in the sequential order that they will appear in the buffer at the destination program; and
3. When finished marshalling all data elements, marshalling the trailer (if present). For ease of use, the marshalling routines preferably provide two buffers, a source buffer for holding the source localized data element, and another buffer, the target buffer, for holding the marshalled data elements.

The "unmarshalling" routines of the invention will now be described. Unmarshalling is the process by which data in the machine-independent CDR format is converted into some target machine and language format. The type of information used by the unmarshalling routines is of two types. The first type is information that can change with the use of the data such as signed/unsigned, packed/not packed, and the like. This information will be passed with the data in the CDR data stream as type attributes. The other type of information is stable information such as data alignment and byte/word sizes. This information may fit naturally into static tables.

As each CDR data element is being read, it is unmarshalled using unmarshalling routines which are built into a library called the Unmarshalling Routines Library (URL). As with the MRLs, each URL will be customized for a particular machine architecture. Thus, there will be one URL version per machine architecture type supported. Much of this customization can be handled through changes in values as the Library is built. As with the MRL, the URL is not designed to be a run time shared library. Rather, the library code will be linked into the INM's code at link time.

A selective low-level routine will be invoked to unmarshall a corresponding data element. The decision of which low-level routine to invoke will be based on what type of CDR data is in the CDR data stream. Thus, there must be a main routine built on top of these low-level routines that will look at the next data element in the CDR stream and use the type attribute ID of that element to select the low-level unmarshalling routine to invoke. Such a routine is straightforward and may be readily designed by one skilled in the art.

Language-specific information about data type sizes and alignments and pointers to the conversion routines to invoke for the DDL type at the destination node are organized into a 2 x 2 matrix called a Language Table. There is one Language Table per language per machine type. Preferably, the rows are DDL types, while the columns are the related localized data element size, alignment and pointer to the specific DDL unmarshalling routine to call:

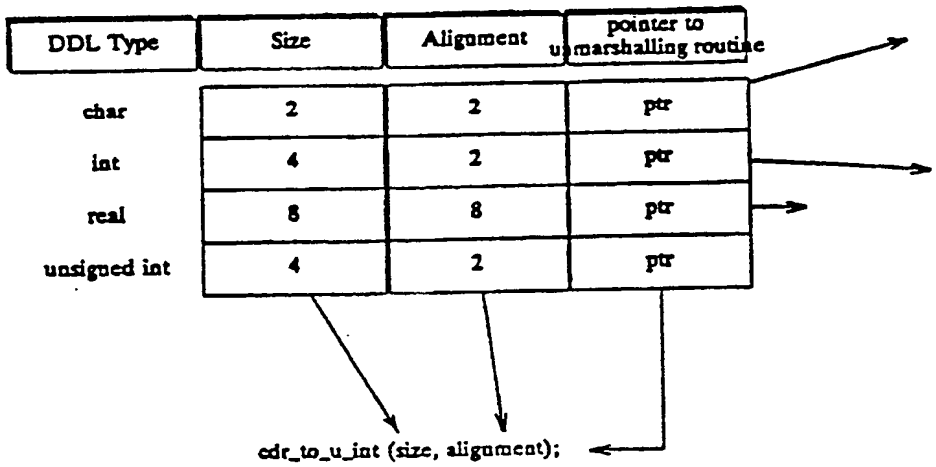


TABLE 2

In order to determine which Language Table to choose upon receipt of data, the header of the CDR is examined for a byte of information that selects what the target language will be. This byte is used to set a global pointer to point to the Language Table to be used to unmarshall that message. To go from this byte value to a global pointer can be done with the following table of pointers to Language Tables:

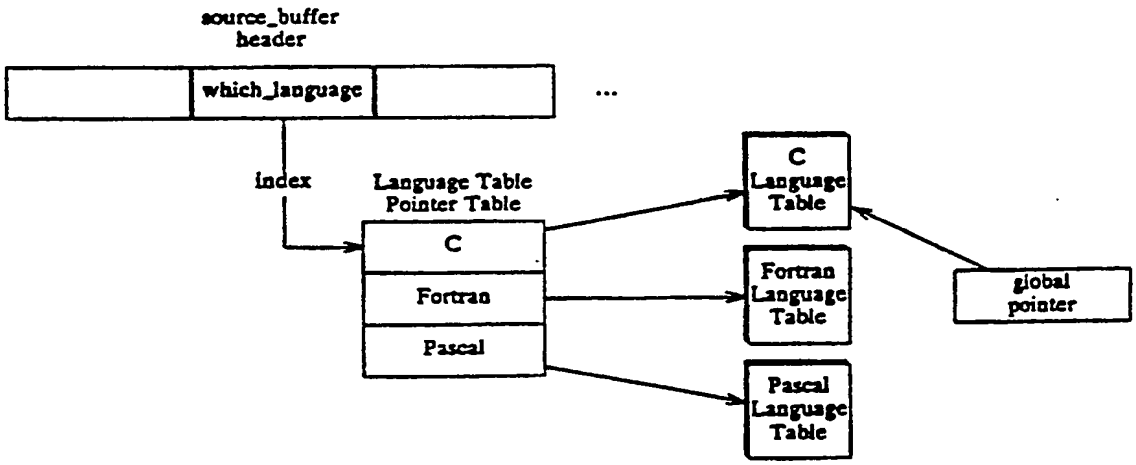


TABLE 3

The Language Table Pointer Table (ltp__table) is set up when the unmarshalling code starts up. During the start of unmarshalling a message, the target language byte value is used as an index into ltp__table to retrieve a pointer to a Language Table. A global pointer is set to point to this Language Table, which is used by the unmarshalling code to convert the remainder of the message.

It would be easy enough to have a global pointer pointing to the selected Language Table. However, a problem arises with Pascal and packed data types. Namely, a CDR of a message arriving on a system with a target language of Pascal may have a mix of packed and unpacked data elements. Each packed versus unpacked data type requires its own size and alignment information in the Pascal Language Table. Thus, the Pascal Language Table would have to contain double the amount of information of the other Language

Tables. This would also require special checking to handle packed data elements since column indexes into the Pascal Language Table would have to be different to access the extra information for packed sizes/alignments for data elements. Since it is desired to minimize the amount of special handling and checking for Pascal packed types, the packed Pascal may be treated as if it resided in another Language Table. Then, by providing a semi-transparent mechanism for dynamically accessing between either the packed or unpacked versions of the Pascal Language Table, the code will be able to run much smoother and faster. The existence of the extra packed Pascal Language Table should also not affect how other languages' Language Tables are accessed. The ltp_table proposed looks like the following:

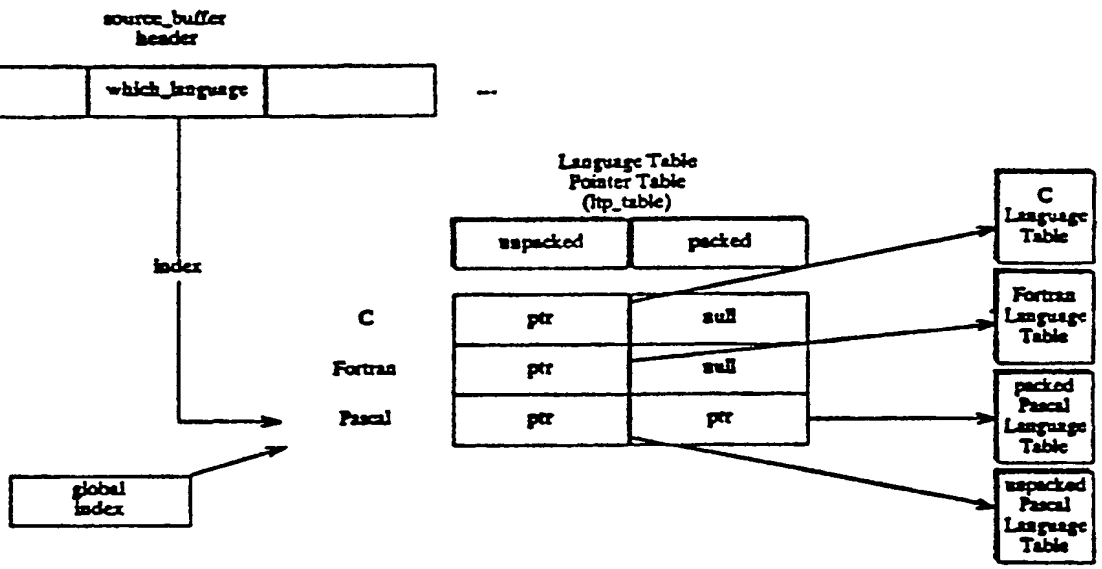


TABLE 4

The byte out of the header of the source_buffer is used as an index into the ltp_table as shown in Table 4. This index is saved in a global variable; therefore, with this index, one knows which target language will be unmarshalled to. Also, the entry the index selects contains two pointers-- one to a packed version of the Language Table, and another to an unpacked version of the Language Table. This only applies to the Pascal entry in ltp_table; the other languages have a NULL pointer for the packed pointer.

When the unmarshalling process comes across a packed array/record type attribute ID (Pascal must be the target language), this type attribute is immediately placed in a stack frame. The rest of the fields in the stack frame are initialized accordingly from information accompanying the type attribute ID. The unmarshalling process continues parsing by looking at the next data element in the input buffer.

However, the data element's size, alignment, and pointer to the conversion routine to invoke must also be determined, for although an index to the ltp_table is available, the question becomes which pointer (column) should be used in that row. This decision depends on whether the previous stack frame was packed or not. The corresponding value is used as the index into the selected row of the ltp_table. The type attribute ID is used as an index into the Language Table to pick up at a particular row the parameters and pointer to the routine to invoke.

As noted above, the user data in the transmitted message is encoded using ASN.1 BER. This is not enough for the receiving side to unmarshall this data into target language data types, for the type attributes and data headers are also marshalled although needed to help in the description. In other words, since all parts of the message follow the ASN.1 Basic Encoding Rules, the other parts must be decoded without the benefit of the data buffer header and type attributes. This is handled by expecting the data buffer header fields to be in a particular order for the data buffer header. Thus, by appropriately placing the data fields, ASN.1 unmarshalling routines may be called to unmarshall the data containing the version and target language data. However, the order of the type attributes field cannot be predicted. Accordingly, the type attributes may be unmarshalled on-the-fly during run time as needed.

Since homogenous data is not marshalled in accordance with the invention, INM 418 must have some

mechanism for determining whether a received message has been converted to the CDR. Generally, this may be accomplished in accordance with the invention by providing a flag in the received data which indicates whether marshalling has been conducted at the sending side. Preferably, the following data transmission sequence is used to ensure that INM 418 will recognize whether the received data is in CDR form.

In particular, on the sending side data from the source application program gets manipulated and marshalled by DMM 412. DMM 412 sets a flag bit to indicate that the dynamic part of the message has been marshalled. DMM then sends the message to the ONM 416 on the local node. ONM 416 places a field containing the size of the total message in front of the message to be sent. The message is then sent to the destination node.

On the receiving side, INM 418 receives the incoming messages and first reads from the network the total size of a message being received. Then INM reads in the number of bytes specified by the total size to obtain the entire message. At the next step, INM 418 looks at the flag field of the message header, and if the "marshall" flag bit in the flag field is set, then the dynamic part of the message is unmarshalled into a new buffer using a special unmarshalling routine. If the "marshall" flag bit is not set, however, this step is bypassed and the received data is sent directly to the input queue of the request manager 406. In this manner, the request manger 406 will be certain to receive data having the proper formatting for the destination application.

SAMPLE CONFIGURATION

As noted above, before one can configure an integration system in accordance with the invention, one must have complete understanding of the system requirements, the system and application capabilities, and the integration functions required by the applications. Such information includes (1) the applications that will be served by the integration system, (2) the data flow (linkage requirements) between those applications, (3) the machine types running the applications, and (4) the type of networking used to tie the systems together. This information must be supplied to configure the integration system in the manner set forth above.

The following example describes the integration tasks using one portion of a computer-integrated manufacturing system which manages work orders. The example will be described with respect to FIGS 8 and 9, where FIG. 8 shows the system to be integrated and FIG. 9 shows the resulting configuration files. In the example, there are three applications handling work orders: a material resource planner (MRP) which generates work orders and operates on an HP9000/S800 computer (CPU1); a scheduler (SCH) which schedules the work and operates on a HP9000/S800 computer (CPU2); and a cell controller (CTL) which which manages the work and reports work completed and operates on an HP90000/S3000 computer (CPU3). These applications are connected over a LAN as shown in FIG. 8. For this example, it will be assumed that the MRP application generates work orders and keeps track of work completed and is written in C programming language, while the SCH application contains information about the machines, people and resources available to work on orders, creates schedules for the work, and is also written in the C programming language. The CTL application manages the work and is written in FORTRAN.

The configuration files for the system are formed as follows:

Network analysis provides information about each of the nodes which must be known before they can be used in the integrated system of the invention. This information is used in the Network Definition File (Netw) to define the nodes in the integration system. As shown in FIG. 9, the network definition file includes a file entry for each node in the integration system on which at least one application runs. In this example, the network definition file preferably describes the following items for each node: its logical name and machine type, whether or not it starts at system start-up, whether or not it is the compilation node, the maximum number of clone processes it can have, its maximum message area size, the maximum percentage of the message area any single process may use, its network type, and its physical node name. For example, application MRP runs on a node having a logical node name CPU1 and a machine type HP90000/S800, is designated to start at start-up, is not a compilation node, and is connected to a LAN type network. Netw contains similar information for the other nodes except that they are compilation nodes.

Data flow analysis of the system is then performed to determine the path that message data takes from a source application to a destination application. In particular, the user determines what each application expects as data, what each application that needs to transmit data produces as data, and how the data is represented and accessed in each application. The portions of the data that will be sent between applications is also determined. In the example, the format and type of data that is exchanged by the three applications in the work order management system are determined. It will be assumed that the applications

exchange data in both files and message packets.

Because the data in the example is exchanged between applications written in different languages, the user is required to define the types of the data when the system is configured. These data descriptions are formed in the Data Definition File (Ddef) which, as noted above, enables messages to be in different formats or applications to be transformed in different languages at the source application and at the destination application to be transformed by using the data definitions in the Ddef file and the manipulations in the Dman file. On the other hand, if the data is in file format, the user can specify the options of the file transfer in a File Definition File (Fdef).

By way of example, application MRP may transfer a work order file to application SCH containing a work order number, a parts number, quantity information and due dates. The application MRP may also receive completion messages from application CTL after manufacturing is complete. Also, since the destination message layout is different from that of the source, a data manipulation must be defined with this message transfer.

Application SCH may transfer instructions in a file for scheduling work from the MRP program and dispatch a message to the CTL application containing: part number, quantity information, start date, start time, stop date, stop time and work order number. Once again, because the destination message layout is different from that of the source, a data manipulation must be defined for this message transfer. Also, completion messages from the CTL program after manufacturing is complete may be received by the application SCH. Moreover, since the source and destination applications are in different languages, a link to correct for representational differences between the languages must be identified.

The CTL application receives dispatched messages from application SCH including: the work order number, the part number, the quantity information, the start data and the start time. Work order completion messages may then be sent to the application MRP, such messages including: work order number, quantity information, part number, completion date and quality information. Also, a work order completion message containing the work order number, the quantity, the part number and completion date may be sent to application SCH.

At the data flow level, the number of processes the system data transfers require, or the specific tasks of those processes varies from one design to another. However, the information gathering procedures and mapping to configuration remain the same. For the example, the process configuration files are shown in FIG. 9 where the MRP application requires a process MRP01CP to send out the work order file and a process MRP02CP to receive completion information. The SCH application also requires two processes: SCH01CP to receive the work order file and SCH02CP to send out a dispatch message and receive a work order completion message. The CTL application, on the other hand, requires only one process: CTL01FP to receive the dispatch message and send out completion messages. Thus, five processes are necessary to transfer the data between the three applications, and two manipulations (MANIP1 and MANIP3) are required because the layout of the data at a destination application is different from the data layout at a source application due to language and machine differences. The process information is stored in the Proc file, while the manipulation information is stored in the Dman file as shown in FIG. 9.

Since the work order management example given primarily exchanges messages, the file definition configuration file (Fdef) contains only FILEDEF1 for the case in which the MRP application sends a work order file to the SCH application. When the file reaches the SCH application, it will replace the existing target file.

The next step in the integration process requires that the language and structural details of the application data be identified. Once the data structure of both the source and destination applications have been identified, the user can decide on the manipulations required to transform the source data structure to the destination structure and data types. This data information goes into the data definition file (Ddef) and data manipulation file (Dman). Thus, the user must know the sizes of the exchanged data so that the corresponding DDL data types may be used to define the data in the Ddef file. The following Table 5 shows how data sizes in the dispatched message are used to correspond to data types between C and DDL and between FORTRAN and DDL:

Source				Destination		
Language: C Sending Buffer: Dispatch message at SCHED				Language: FORTRAN Receiving Buffer: Dispatch message at CTL		
C structure		DDL data definition Name: disp_c		FORTRAN	DDL data definition Name: disp_f	
char	part_no[17]	part_no	: string[17]		part_no	: string[16]
int	qty	qty	: integer	** *See below*	wo_no	: string[10]
char	start_date[7]	start_date	: string[7]	*for FORTRAN*	qty	: integer
char	start_time[5]	start_time	: string[5]	*definitions*	start_date	: string[6]
char	stop_date[7]	stop_date	: string[7]		start_time	: string[4]
char	stop_time[5]	stop_time	: string[5]		stop_date	: string[6]
char	wo_no[11]	wo_no	: string[11]		stop_time	: string[4]

TABLE 5

The information from Table 5 may be used to determine the manipulation pattern necessary to transform the data in the integration system. These manipulations go into the data manipulation file Dman. For the present example, the manipulation statement that is required is a MOVE of the source structure (Disp__c) to the destination structure (Disp__f) (i.e., MOVE disp__c TO disp__f). This manipulation statement is contained in the example manipulation called Manip1.

The user is now ready to describe the nodes where the source data defined in the DDL or DML can originate. This information is stored in the Link Definition File (Link). The links bind the source and destination language types to a specific data definition or data manipulation at validation. At run time, the Link file designates the distribution of node-specific data manipulations to the appropriate nodes. For example, Link1 requires Manip1 to convert from source language C to destination language FORTRAN for a message originating at source node CPU2 and destined for node CPU3.

For the example given in FIGS. 8 and 9, the Link file defines the four links required for the work order management system example. Link1 binds Manip1 (the manipulation performed on the dispatch message sent from SCH to CTL) to the C language used by the SCH application. Link2 on the other hand, corrects for representational differences in the work order complete message sent from the FORTRAN application CTL to the C application SCH. The message has the same record format but is in a different language. Link2 thus points to the DDL definition of the source data structure (wo__comp__f) in the Ddef file. Link3 binds Manip3 (the manipulation performed on the work order complete message sent from CTL to MRP) to the FORTRAN language used by the CTL application. Finally, Link1 defines the link for a file transfer. The file transfered, filedef1, is the work order file that the MRP application sends to the SCH application. The NULL value indicates that no manipulations are required, only specification of destination file attributes. CPU2 and CPU3 receive the DMM modules (at startup) that enable them to handle the transfers.

FIG. 9 shows the configuration files as created for the work order management system example described above and shows how information defined in each file is referenced by the other files. This configuration file set is validated before it is used in startup in the manner described above with respect to FIG. 5. Thus, DMM 412 is created specifically to handle the manipulation specified by the user in the configuration files. For this purpose, these files are validated before startup and loaded (at startup) on each node as Configuration Tables as previously described.

Although an exemplary embodiment of the invention has been described in detail above, those skilled in the art will readily appreciate that many additional modifications are possible in the exemplary embodiment without materially departing from the novel teachings and advantages of the invention. For example, access to a remote database using the invention may be possible by eliminating the dependencies on a specific set of database management system calls. Thus, an application accessing another application's database through the remote data access routines will be freed from changes should the database structure or the

database management system used by the target application change. This can be accomplished through the syntactical and structural definition of the target application database in the configuration files. In addition, the configuration files of the invention may have domains, where each domain contains a configuration for a system running on a physical network. There can thus be several domains operating concurrently on any physical network, and these domains can be managed by the user in accordance with the invention by configuring the nodes into non-overlapping domains. Also, the invention may automatically determine the optimal allocation for data manipulation in order to balance the system load. Accordingly, these and all such modifications are intended to be included within the scope of this invention as defined in the following claims.

APPENDIX A

The following routines are used for initializing or removing communication to the system of the invention:

- SpInit: Used to initiate communication between the system and the calling process.
- SpEbd: Used to end communication and cleanup resources allocated by the system for the calling process.

The following function is used to transfer messages between processes:

- SpSendMsg: Used to send a message to a process.

The following function is used for transferring files:

- SpSendFile: Used to copy or append a file to a destination file.

The following functions are used to control system processes:

- SpStartProcess: Used to start a system process.
- SpStopProcess: Used to stop a system process.

The following function is used to read incoming messages:

- SpReadQ: Used to read messages from the process' incoming message queue.

The following function is used to control user access routine options:

-SpControl: Used to set user-controlled options for the system access routines.

The following function is used to allow the user to access the system error logger:

- SpErrLog: Used to allow the user to send application-specific messages to the system error logger.

The following function is used to clear the application queue. This function should be used with care.

- SpFlush: Used to set user-controlled options for the system access routines.

Access Routine Examples

SpInit - Every process must do an SpInit. In this example, SpInit initiates communication between the integration system and the process named mrp01cp.

```
int   rc;
int   i;
int   in_list[5], out_list[5];
int   flags;
char  srcprocln[16];
      .
      .
      .
for (i = 0; i < 5; i++)
{
      in_list[i] = 0;
      out_list[i] = 0;
}
flags = 0;
strcpy (srcprocln, "mrp01cp");

if ((rc = SpInit(flags, srcprocln, in_list, out_list)) !=0)
{
      printf ("ERROR: SpInit of %s returns %d\n", srcprocln, rc);
      exit(1);
}
else printf ("SpInit was successful...\n", rc);
```

SpEnd - SpEnd ends communication between the integration system and a process.

```
int   rc;
int   flags;
      .
      .
      .
flags = 0;
if ((rc = SpEnd(flags)) !=0)
      printf ("ERROR: SpEnd returns %d\n", rc);
else
      printf ("SpEnd was successful...\n", rc);
```

SpSendMsg In this example, SpSendMsg sends a message to the message buffer called SCHWOFILE in the process named sch01cp.

```
int                rc;
int                i;
int                in_list[5], out_list[5];
int                flags;
char               desprocln[16];
unsigned char      dfilename[MAXBUFSIZE];
char               linkname[16];
      .
      .
      .
for (i = 0; i < 5; i++)
{
      in_list[1] = 0;
      out_list[1] = 0;
}
flags = 0;
strcpy (desprocln, "sch01cp");
strcpy (dfilename, SCHWOFILE);
strcpy (linkname, "");
in_list[SpBUF_LEN] = strlen(dfilename) +1;
in_list[SpTAG] = 1;

if ((rc = SpSendMsg(flags, desprocln,dfilename, linkname, in_list,
            out_list)) !=0)
      printf("ERROR: SpSendMsg returns %d\n", rc);
else
      printf("SpSendMsg successful...\n");
```

SpSendFile In this example, SpSendFile sends the source file MRPWOFILE to the LOGINFO file in the process named sch01cp. It uses the file attributes specified in the link file linkf1.

```
int                 rc;
int                 i;
int                 in_list[5], out_list[5];
int                 flags;
char                desprocln[16];
unsigned char       srcfile[MAXBUFSIZE];
unsigned char       dfilename[MAXBUFSIZE];
unsigned char       desfile[MAXBUFSIZE];
char                linkname[16];
      .
      .
      .
for (i = 0; i < 5; i++)
{
      in_list[1] = 0;
      out_list[1] = 0;
}
flags = 0;
strcpy (desprocln, "sch01cp");
strcpy (srcfile, MRPWOFILE);
strcpy (dfilename, SCHWOFILE);
strcpy (desfile, LOGINFO);
strcat (desfile, dfilename);
strcpy (linkname, "linkf1");

if ((rc = SpSendFile(flags, desprocln, srcfile, desfile, linkname, in_list,
            out_list)) !=0)
      printf("ERROR: SpSendFile returns %d\n", rc);
else
      printf("SpSendFile successful...\n");
```

SpStartProcess In this example, SpStartProcess starts the process named sch01cp.

```
int   rc;
int   i;
int   in_list[5], out_list[5];
int   flags;
char  desprocln[16];
char  runstring[128];
char  execinfo[256];
      .
      .
      .
for (i = 0; i < 5; i++)
{
      in_list[i] = 0;
      out_list[i] = 0;
}
flags = 0;
strcpy (desprocln, "sch01cp");
strcpy (runstring, "");
strcpy (execinfo, "");
in_list[SpPASSWD_NUM] = 100;

if ((rc = SpStartProcess(flags, desprocln, runstring, execinfo, in_list,
            out_list)) !=0)
      printf("ERROR: SpStartProcess returns %d\n", rc);
else
      printf("SpStartProcess successful...\n");
```

SpStopProcess

In this example, SpStopProcess stops a process named "test02" when the calling program receives the SIGUSR1 interrupt. The password number in_list parameter must match the password number specified for this process in the Process Definition configuration file to stop this process.

```
int   flags;
char  desprocln[16]
int   in_list[5], out_list[5];
      .
      .
      .
flags = 0;
strcpy(desprocln, "test02");
in_list[SpPASSWD_NUM] = 100;
in_list[SpSTOP_SIG] = SIGUSR1;

if ((rc = SpStopProcess(flags, desprocln, in_list, out_list)) !=0)
{
      printf("ERROR: SpStopProcess returns %d\n", rc);
      exit(1);
}
else
{
      printf("SpStopProcess was successful.\n");
}
```

SpReadQ In this example, SpReadQ reads the incoming message area and places the message with the tag value of 1 in the buffer area.

```
int                rc;
int                i;
int                in_list[5], out_list[5];
int                flags;
char               procln[16];
unsigned char      buffer[MAXBUFSIZE];
      .
      .
      .
for (i = 0; i < 5; i++)
{
      in_list[i] = 0;
      out_list[i] = 0;
}
flags = 0;
in_list[SpBUF_LEN] = MAXBUFSIZE;
in_list[SpTIMEOUT] = 0;
in_list[SpLOWER_TAG] = 1;
in_list[SpUPPER_TAG] = 0;

if ((rc = SpReadQ(flags,procln, buffer, in_list, out_list)) !=0)
      printf("ERROR: SpReadQ returns %d\n", rc);
else
      printf("SpReadQ successful...\n");
```

spControl

In this example, the SpControl function parameter SpSET_ FILE_ NOTIF notifies the calling program when a file sent through the system is available to it. The calling program is notified by means of the interrupt SIGUSR1 specified in the in_list parameter.

```
int             func;
int             in_list[5], out_list[5];
int             i;

void     handler_routine( );
static struct    sigvec sig_hdlr = (handler_routine, 0, 0);

/* Set up a handler for SIGUSR1 signal. */
sigvector(SIGUSR1,  &sig_hdlr, 0);
 .
 .
 .
func = SpSET_FILE_NOTIF;
for (i = 0; i < 5; i++)
{
 in_list[i] = 0;
 out_list[i] = 0;
}

in_list[0] = 1;
in_list[1] = SIGUSR1;
if ((rc = SpControl(func, NULL, in_list, out_list)) !=0)
{
 printf("ERROR: SpControl of %s returns %d\n", procln, rc);
 exit(1);
}
else
{
 printf("SpControl was successful...\n", rc);
}
 .
 .
void     handler_routine( )
{
 printf("In handler routine ...\n");
}
```

SpErrLog

This example shows using SpErrLog to include a user-written error message in the system's error logger.

```
char errmsg[MAXMSGSIZE];
int  errmsglen;
     .
     .
     .
strcpy(errmsg, "User Error Msg: From recvstop process at step1.");
errmsglen = strlen(errmsg);

if ((rc = SpErrLog(errmsg, errmsglen)) !=0)
     printf("ERROR: SpErrLog returns %d\n", rc);
else
     printf("SpErrLog was successful...\n");
```

SpError

In this example, SpError retrieves an ASCII message buffer corresponding to a system error number.

```
int  rc1;
int  rc;
char errmsg[MAXMSGSIZE];
     .
     .
     .
if ((rc1 = SpError(rc, errmsg)) != 0)
     printf("ERROR: SpError returns %d\n", rc1);
else
     printf("Error msg is %s", errmsg);
exit(1);
```

SpFlush SpFlush flushes pending request and clears the timeout list for the process. Use it only if your adaptor uses an interrupt-handling routine that needs to do a longjmp when it is called.

```
jmpbuf      signal_jmpbuf;
      .
      .
      .
setjmp(signal_jmpbuf);

      .
      .
      .
/* This section of the program describes an infinite wait using    */
/* the SpReadQ access routine to read incoming messages.           */
      .
      .
      .

/* The interrupt handler */
void interrupt_handler( )
{
      extern jmp_buf signal_jmpbuf;

      .
      .
      .
      printf("In signal handler function.\n");

      int   rc;
      if ((rc = SpFlush( )) != 0)
      {
            printf("ERROR: SpFlush returns %d\n"
}
```

SpDelGuaranteedMsg This is an example of using SpDelGuaranteedMsg to delete a guaranteed message.

```
if ((rc = SpDelGuaranteedMsg( )) !=0)
{
      printf("ERROR: SpDelGuaranteedMsg returns %d\n", rc);
}
else
      printf("SpDelGuaranteedMsg successful...\n");
```

APPENDIX B

Example Adaptor Programs

This is a brief description of the functions of the adaptors created for the integrated system example.

mrp01cp — The mrp01cp adaptor initiates the system process mrp01cp which sends a work order file and a message containing the destination file name to the sch01cp process. The mrp01cp process also starts up the sch01cp program. Mrp01cp uses these calls: SpInit, SpStartProcess, SpSendFile, SpSendMsg, and SpEnd.

sch01cp — The sch01cp initates the system process sch01cp that uses a SpReadQ call to get the name of the work order file sent from mrp01cp. Sch01cp opens the work order file and generates a new dispatch file called SCHDISPF. Sch01cp uses these calls SpInit, SpReadQ, and SpEnd. It sets the flat SpDELETE_ALL with the SpInit to delete all messages that have been stored for this process.

sch02cp — The sch02cp adaptor initiates the system process sch02cp. Sch02cp sends messages to process ct101fp and receives messages from ct101fp. Sch02cp uses these calls: SpInit, SpSendMsg, and SpEnd.

ct101fp — The ct101fp adaptor in FORTRAN initiates the system process ct101fp that hangs on an SpReadQ call waiting for a data message from the process sch02cp. Ct101fp processes the data in the message and returns to wait for another SpReadQ call. The ct101fp process also sends messages to two other processes: sch02cp and mrp02cp.

ct101pp — The ct101 pp adaptor is a passed version of ct101 fp.

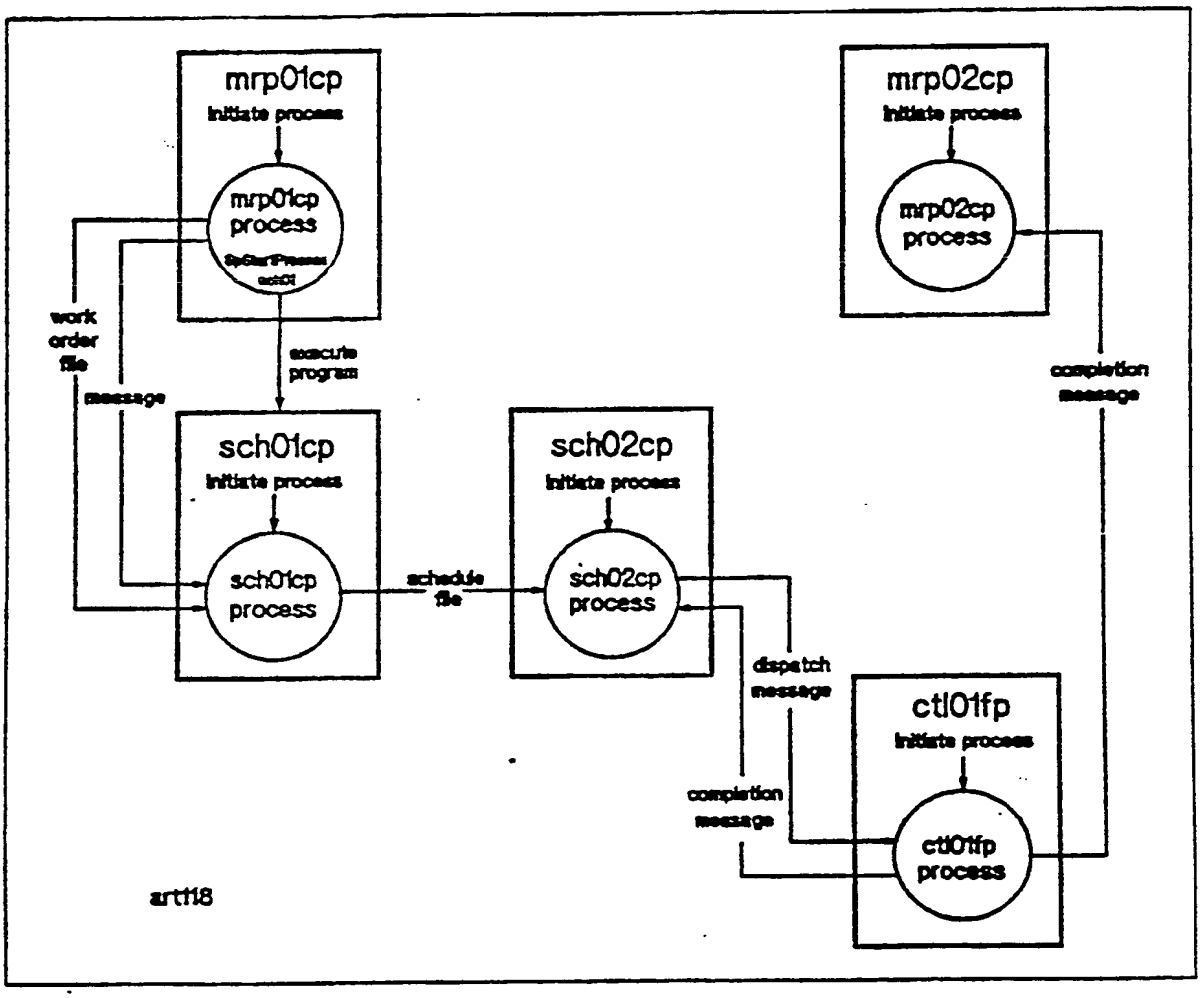
mrp02cp — The mrp02cp adaptor initiates the background process mrp02cp that hands on an SpReadQ call waiting for a data message from the process ct101fp. Mrp02cp uses these calls:
SpInit, SpReadQ, and SpEnd.

These are additional programs created for this work order management example:

ex01.h — This is a header file that defines the file directories and structures for the adaptors used in this system example.

datagen — This is a program to solicit user input for creating a work order database.

Adaptor Examples



Example Application Adaptors and Processes

mrp01cp

```
/*****************************************************************************/
/*  $Source:  mrp01cp.c                                                      */
/*  $Revision:  1.1 $                                                        */
/*                                                                           */
/*  This is a skeleton application adaptor to send a file                    */
/*  to another HP Sockets adaptor.  The mrp01cp adaptor initiates the mrp01cp */
/*  process which sends a work order file and a message containing the       */
/*  destination file name to the sch01cp process.  The mrp01cp process also  */
/*  executes the sch01cp program.  Mrp01cp uses these calls: SpInit,         */
/*  SpStartProcess, SpSendFile, SpSendMsg, and SpEnd.                        */
/*****************************************************************************/

/* ------------------------------------------------------
 This is the maximum message size HP Sockets allows.
 Change it to the size that the application needs.
 ------------------------------------------------------ */

#define MAXBUFSIZE 1024
#define MAXMSGSIZE 500

/* Global include files */

#include <stdio.h>
#include <Sp.c.h>        /* HP Sockets header include file for C programs    */
#include "ex01.h"        /* Data and file definitions for work order management
                            examples                                         */

/* ===== Main program ===== */

main( )
{
/* ------------------------------------------------------------------------
 These are suggested parameter definitions for all access routine calls.
 ------------------------------------------------------------------------ */

 int             rc;                      /* Access routine return code   */
 int             buflen;                  /* Data buffer length           */
 unsigned char   buffer[MAXBUFSIZE];      /* Data buffer received         */
 char            procIn[16];              /* Process name initiate        */
```

```
        char            srcprocln[16];          /* Source process name          */
        char            desprocln[16];          /* Destination process name     */
        char            linkname[16];           /* Destination process name     */

        char            runstring[128];         /* Runstring for SpStartProcess */
        char            execinfo[256];          /* Execution information for
                                                        SpStartProcess          */

        int             flags;                  /* Flag parameter               */
        int             func;                   /* Set function for SpControl   */
        char            errmsg[MAXMSGSIZE];     /* Return error buffer          */
        int             errmsglen;              /* Set error message length     */
        int             in_list[5],             /* Additional input parameters  */
                        out_list[5];            /* Additional output parameters */

    /* ====== These are special parameters used in  this program. ====== */

        int             i;
        unsigned char   srcfile[MAXBUFSIZE];    /* Source file name             */
        unsigned char   desfile[MAXBUFSIZE];    /* Destination file name        */
        unsigned char   dfilename[MAXBUFSIZE];  /* Destination file name only   */

    /* ------------------------------------------------------------
     Start communications with HP Sockets.  Set up the parameters and
     make the SpInit call.
    ------------------------------------------------------------ */
    for (i = 0; i < 5; i++)
    {
     in_list[i] = 0;
     out_list[i] = 0;
    }
    flags = 0;
    strcpy (srcprocln, "mrp01cp");

    if ((rc = SpInit(flags, srcprocln, in_list, out_list)) != 0)
    {
        printf("ERROR: SpInit of %s returns %d\n", srcprocln, rc);
        exit(1);
    }
    else
        printf("SpInit was successful...\n", rc);
```

```
/* ------------------------------------------------------------
 After each call, reset the parameters.  Make the SpStartProcess
 call to execute sch01cp.
------------------------------------------------------------ */

for (i = 0; i < 5; i++)
{
 in_list[i] = 0;
 out_list[i] = 0;
}

flags = 0;
strcpy(desprocln, "sch01cp");
strcpy(runstring, "");          /* Use default runstring in proc.def        */
strcpy(execinfo, "");           /* Use default execution info in proc.def   */
in_list[SpPASSWD_NUM] = 100;    /* Password number 100 was defined in Proc.def*/

if ((rc = SpStartProcess(flags, desprocln, runstring, execinfo,
                         in_list, out_list)) != 0)
    printf("ERROR: SpStartProcess returns %d\n", rc);
else
    printf("SpStartProcess successful...\n");

/* ------------------------------------------------------------
 Send file with wait.  We use the 60 second default timeout.
 If you need a longer timeout value, use SpControl to
 set up the timeout value.
------------------------------------------------------------ */

for (i = 0; i < 5; i++)
{
 in_list[i] = 0;
 out_list[i] = 0;
}

flags = 0;
strcpy(desprocln, "sch01cp");

strcpy(srcfile, MRPWOFILE);
strcpy(dfilename, SCHWOFILE);
strcpy(desfile, LOGINFO);
strcat(desfile, dfilename);
```

```
strcpy(linkname, "linkfl");

if ((rc = SpSendFile(flags, desprocln, srcfile, desfile, linkname,
                     in_list, out_list)) != 0)
    printf("ERROR: SpSendFile returns %d\n", rc);
else
    printf("SpSendFile successful...\n");

/* ------------------------------------------------------------
 After the file has been sent, bundle a message that
 contains the destination file name.  Send this message
 to the destination process.
------------------------------------------------------------ */

for (i = 0; i < 5; i++)
{
 in_list[i] = 0;
 out_list[i] = 0;
}

flags = 0;
strcpy(desprocln, "sch01cp");

strcpy(srcfile, MRPWOFILE);
strcpy(dfilename, SCHWOFILE);

strcpy(linkname, "");
in_list[SpBUF_LEN] = strlen(dfilename) + 1;
in_list[SpTAG] = 1;
if ((rc = SpSendMsg(flags, desprocln, dfilename, linkname, in_list,
                    out_list)) !=0)
{
    printf("ERROR: SpSendMsg returns %d\n", rc);
}
else
    printf("SpSendMsg successful...\n");

/* ----- End communications with HP Sockets ----- */

flags = 0;

if ((rc = SpEnd(flags)) !=0)
    printf ("ERROR: SpEnd returns %d\n", rc);


        else
            printf("SpEnd was successful...\n", rc);
        }
```

sch01cp

```
/*****************************************************************************/
/*  $Source:     sch01cp.c                                                   */
/*  $Revision:   1.1 $                                                       */
/*                                                                           */
/*  This program initiates the sch01cp process that uses a SpReadQ call     */
/*  to get the file name of the work order file sent from the mrp01cp process. */
/*  It opens this work order file and generates a new dispatch file          */
/*  called SCHDISPF.  Sch01cp uses these calls: SpInit, SpReadQ,             */
/*  and SpEnd.  It sets the flag SpDELETE_ALL with SpInit to                 */
/*  delete all messages that have been stored for this process.              */
/*                                                                           */
/*****************************************************************************/

/* ------------------------------------------------------------
 This is the maximum buffer size allowed by HP Sockets.
 Change it to the size that the application needs.
------------------------------------------------------------ */

#define MAXBUFSIZE 1024
#define MAXMSGSIZE 500

/* Global include files  */

#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <Sp.c.h>        /* HP Sockets header include file for C programs    */
#include "ex01.h"        /* Data and file definitions for work order management
                            examples                                         */

/*****************************************************************/
/*  Function:    process_file                                    */
/*  Parameter:   filename                                        */
/*  Return:      normal = 0, error = -1                          */
/*                                                               */
/*  Description: This function reads a work order file           */
/*               and generates a dispatch file.                  */
/*                                                               */
/*               argument        meaning                         */
```

```
/*              --------        -------                                 */
/*              filename        name of the work order file             */
/*                              on the SCHED application                */
/*                                                                      */
/************************************************************************/

int     process_file(filename)
char    *filename;

{
 FILE   *wo_fp,                 /* file descriptor */
        *disp_fp;               /* file descriptor */

 wo_rec         *wo_ptr, wo_buf;
 disp_rec       *disp_ptr, disp_buf;

 int    i;

 long cur_time;                 /* The current time(seconds since Jan 1, 1970)*/
 char *char_str_ptr;
 char time_str[5];
 char date_str[7];

/* ===== Get the system's time. ====== */
 cur_time = time((long *) 0);   /* Get the system's time. */
 char_str_ptr = nl_cxtime(&cur_time, "%y%m%d");
 strncpy(date_str, char_str_ptr, 7);

 disp_ptr = &disp_buf;
 wo_ptr = &wo_buf;

/* ====== Open the work order file for a read ====== */

 if( (wo_fp = fopen (filename, "r")) == NULL)
 {
    printf("Could not open work order file %s.\n", filename);
    return(-1);
 }
/* ===== Open the dispatch file for a write ===== */

 else if ( (disp_fp = fopen(SCHDISPF, "w")) == NULL)
 {
    printf("Could not open dispatcher file %s.\n", SCHDISPF);
```

```
      return(-1);
   }

/* ------------------------------------------------------------------------
 Simulate a schedule system application that generates dispatch information.
 ------------------------------------------------------------------------ */

 while ((i = fread(wo_ptr, sizeof(wo_rec), 1, wo_fp)) == 1)
 {
    printf("In fread loop.\n");
    strcpy(disp_buf.wo_no, wo_ptr->wo_no);
    disp_buf.qty = wo_ptr->qty;
    strcpy(disp_buf.part_no, wo_ptr->part_no);
    strcpy(disp_buf.start_date, date_str);
    strcpy(disp_buf.start_time, "0800");
    strcpy(disp_buf.stop_date, date_str);
    strcpy(disp_buf.stop_time, "0900");
    if((i = fwrite(disp_ptr, sizeof(disp_rec), 1, disp_fp) ) != 1)
    {
          printf("fwrite error for %s.\n", SCHDISPF);
          return(-1);
    }
 }

 fclose(wo_fp);                              /* Close work order file  */
 fclose(disp_fp);                            /* Close dispatch file    */

}

/* ===== Main program ====== */

main()
{
/* ------------------------------------------------------------------------
 These are suggested parameter definitions for all access routine calls.
 ------------------------------------------------------------------------ */

 int            rc;                          /* Access routine return code   */
 int            buflen;                      /* Data buffer length           */
 unsigned char  buffer[MAXBUFSIZE];          /* Data buffer received         */
 char           procln[16];                  /* Process name initiate        */
 char           srcprocln[16];               /* Source process name          */
 char           desprocln[16];               /* Destination process name     */
```

```
char            linkname[16];           /* Destination process name     */

char            runstring[128];         /* Runstring for SpStartProcess */
char            execinfo[256];          /* Execution information for
                                                SpStartProcess           */

int             flags;                  /* Flag parameter               */
int             func;                   /* Set function for SpControl   */
char            errmsg[MAXMSGSIZE];     /* Return error buffer          */
int             errmsglen;              /* Set error message length     */
int             in_list[5],             /* Additional input parameters  */
                out_list[5];            /* Additional output parameters */

/* ------ These are special parameters used in  this program. ------*/

int             i;
int             still_have_work = 1;    /* True for do loop             */


/* ------------------------------------------------------------------
 Start communications with HP Sockets.  Set up the parameters and
 make the SpInit call.
 ------------------------------------------------------------------ */

 for (i = 0; i < 5; i++)
 {
   in_list[i] = 0;
   out_list[i] = 0;
 }
 flags = SpDELETE_ALL;
 strcpy(srcprocln, "sch01cp");

 if ((rc = SpInit(flags, srcprocln, in_list, out_list)) != 0)
 {
    printf("ERROR: SpInit of %s returns %d\n", srcprocln , rc);
    exit(1);
 }
 else
    printf("SpInit was successful...\n", rc);

/* ------------------------------------------------------------------
 Do an SpReadQ call to get the name of the work order file
 in the message sent by the mrp01cp program.
```

```
   ------------------------------------------------------------ */

    for (i = 0; i < 5; i++)
    {
      in_list[i] = 0;
      out_list[i] = 0;
    }

    flags = 0;
    in_list[SpBUF_LEN] = MAXBUFSIZE;
    in_list[SpTIMEOUT] = 0;          /* Set timeout forinfinite wait. */
                                     /* Default is 60 seconds. */

    in_list[SpLOWER_TAG] = 1;
    in_list[SpUPPER_TAG] = 0;

    if ((rc = SpReadQ(flags, procIn, buffer, in_list, out_list))
                    != 0)
       printf("ERROR: SpReadQ returns %d\n", rc);
    else
    {
       printf("File %s received.\n", buffer);
       process_file(buffer);                    /* Call the function defined above */
    }

   /* ----- End communications with HP Sockets ----- */

    flags = 0;
    for (i = 0; i < 5; i++)
    {
      in_list[i] = 0;
      out_list[i] = 0;
    }

    if ((rc = SpEnd(flags)) != 0)
       printf("ERROR: SpEnd returns %d\n", rc);
    else
       printf("SpEnd was successful...\n");
   }
```

sch02cp

```
/*****************************************************************/
/*   Source:     sch02cp.c                                       */
/*  $Revision:   1.1 $                                           */
/*                                                               */
/*  This is an application adaptor to send messages to           */
/*  another HP Sockets adaptor.  The sch02cp adaptor initiates   */
/*  the sch02cp process which sends messages to process ctl01fp  */
/*  and receives messages from it.  Sch02cp uses these calls:    */
/*  SpInit, SpSendMsg, and SpEnd.                                */
/*****************************************************************/

/* ============================================================
 This is the maximum buffer size HP Sockets allows.
 Change it to the size that the application needs.
============================================================ */

#define MAXBUFSIZE 1024
#define MAXMSGSIZE 500

/* Global include files  */

#include <stdio.h>

#include <Sp.c.h>        /* HP Sockets header include file for C programs    */
#include "ex01.h"        /* Data and file definitions for work order management
                            - examples                                       */

/* ===== Main program ===== */

main()
{
/* ============================================================================
 These are suggested parameter definitions for all access routine calls.
============================================================================ */

 int            rc;                    /* Access routine return code   */
 int            buflen;                /* Data buffer length           */
 unsigned char  buffer[MAXBUFSIZE];    /* Data buffer received         */
 char           procln[16];            /* Process name initiate        */
```

```
char            srcprocln[16];          /* Source process name          */
char            desprocln[16];          /* Destination process name     */
char            linkname[16];           /* Destination process name     */

char            runstring[128];         /* Runstring for SpStartProcess */
char            execinfo[256];          /* Execution information for
                                                SpStartProcess           */

int             flags;                  /* Flag parameter               */
int             func;                   /* Set function for SpControl   */
char            errmsg[MAXMSGSIZE];     /* Return error buffer          */
int             errmsglen;              /* Set error message length     */
int             in_list[5],             /* Additional input parameters  */
                out_list[5];            /* Additional output parameters */

/* ------These are special parameters used in  this program. ------ */

int             i;
char            contchar,
                dummy[80];

disp_rec        *disp_ptr, disp_buf;
wo_compl_s_rec  *comp_ptr, comp_buf;

FILE            *disp_fp;

disp_ptr = &disp_buf;
comp_ptr = &comp_buf;

/* ----- Open the dispatch file for a read ----- */

if ( (disp_fp = fopen(SCHDISPF, "r")) == NULL)
{
    printf("Could not open dispatcher file %s.\n", SCHDISPF);
    exit(1);
}

/* ------------------------------------------------------------------
 Start communications with HP Sockets.  Set up the parameters and
 make the SpInit call.
 ------------------------------------------------------------------ */

for (i = 0; i < 5; i++)
```

```
    {
      in_list[i] = 0;
      out_list[i] = 0;
    }
    flags = 0;
    strcpy(srcprocln, "sch02cp");

    if ((rc = SpInit(flags, srcprocln, in_list, out_list)) != 0)
    {
       printf("ERROR: SpInit of %s returns %d\n", srcprocln , rc);
       exit(1);
    }
    else
       printf("  SpInit was successful...\n", rc);

   /* ------------------------------------------------------------
    This is the area where the data from the application should
    be accessed (the file is read, the shared memory area is
    accessed, etc).  This portion is application-specific.

    This data should be placed in the buffer variable "buffer",
    and the number of bytes should be put into variable "buflen".
     ------------------------------------------------------------ */

   /* ----- Initialize the continuous flag to yes ----- */

    contchar = 'y';

   /* ----- Read the dispatch record ----- */

    while ( ((i = fread(disp_ptr, sizeof(disp_rec), 1, disp_fp)) == 1)
       && (contchar == 'y') )
    {

      printf("Do you want to send this dispatch record to control system?\n");
      contchar = getchar();
      gets(dummy);
      if (contchar == 'y')
      {
   /* ------------------------------------------------------------------
    Use SpSendMsg to send the dispatcher buffer to a FORTRAN
    controller program.
   ------------------------------------------------------------------ */
```

```
        for(i = 0; i < 5; i++)
        {
             in_list[i] = 0;
             out_list[i] = 0;
        }

        flags = SpNOWAIT_FLAG;
        strcpy(desprocln, "ctl01fp");
        strcpy(linkname, "link1");
        in_list[SpBUF_LEN] = sizeof(disp_rec);
        in_list[SpTAG] = 10;

        if ((rc = SpSendMsg(flags, desprocln, disp_ptr, linkname,
                    in_list, out_list)) != 0)
        {
            printf("ERROR: SpSendMsg of returns %d\n", rc);
        }
        else
        {
            printf("SpSendMsg successful...\n");

    /* ------------------------------------------------------------------
     Do a ReadQ with a long wait to get the work order complete
     message from the cell controller application.
    ------------------------------------------------------------------ */

            for (i = 0; i < 5; i++)
            {
              in_list[i] = 0;
              out_list[i] = 0;
            }
            flags = 0;
            in_list[SpLOWER_TAG] = 0;
            in_list[SpBUF_LEN] = sizeof(comp_buf);
            in_list[SpTIMEOUT] = 0; /* Infinite  wait. Default is 60 sec */

            if ((rc = SpReadQ(flags, procln, &comp_buf, in_list, out_list)) >0)
            {
                    printf("ERROR: SpReadQ of returns %d\n", rc);
            }
            else
            {
```

```
                    printf("SpReadQ successful...\n");
                    printf("The previous work order had been completed.\n");
                    printf("wo_comp_wo_no = %s\n", comp_buf.wo_no);
                    printf("wo_comp_part_no = %s\n", comp_buf.part_no);
                    printf("wo_comp_qty = %d\n", comp_buf.qty);
                }

            }
          }
        }

    /* ----- End communications with HP Sockets ----- */

        if ((rc = SpEnd(flags)) != 0)
           printf("ERROR: SpEnd returns %d\n", rc);
        else
           printf("  SpEnd was successful...\n", rc);

    }
```

ctl01fp

FORTRAN programs on the HP9000/S800 require the LITERAL_ALIAS ON directive shown in this example. This directive specifies that any external names in an ALIAS directive are processed as they appear; that is, they are neither upshifted nor downshifted. The default is OFF and external names are upshifted or downshifted according to the UPPERCASE directive setting. Refer to the *HP FORTRAN 77 Reference Manual (Part number 5957-4685)*.

```
$LITERAL_ALIAS ON
C ************************************************************** C
C                                                                C
C    Source:     ctl01fp.f                                       C
C   $Revision:   1.1 $                                           C
C                                                                C
C   This is a FORTRAN application adaptor that initiates the     C
C   process ctl01fp which hangs on an SpReadQ call waiting for   C
C   a data message from the process sch02cp.  Ctl01fp processes  C
C   the data in the message and returns to wait for another      C
C   SpReadQ call.  The ctl01fp process also sends messages to    C
C   two other processes: sch02cp and mrp02cp.                    C
C ************************************************************** C
          program ctl01fp

$INCLUDE 'Sp.f.h'

      integer           rc
      integer           buflen
      character         buffer(1024)
      character*16      procln
      character*16      srcprocln
      character*16      desprocln
      character*16      linkname

      character*128     runstring
      character*256     execinfo

      integer           flags
      integer           func
      character*256     errmsg
      integer           errmsglen
      integer           timeout
```

```
      integer           ilist(5), olist(5)

C  ****** Declaration section ******

      integer           i

      character*1024    msgbuf

C  Field definitions (dispatch information) for main example

      character*16      part_no
      character*10      wo_no
      integer           qty
      character*6       start_date, stop_date
      character*4       start_time, stop_time

      equivalence (msgbuf,buffer)
      equivalence (buffer(1), part_no), (buffer(17), wo_no),
     +          (buffer(29), qty), (buffer(33), start_date),
     +          (buffer(39), start_time), (buffer(43), stop_date),
     +          (buffer(49), stop_time)

      character         outbuf(38)

C   Work complete information

      character*18      c_part_no
      character*12      c_wo_no
      integer           c_qty
      character*8       c_date_comp

      equivalence (outbuf(1), c_part_no), (outbuf(19), c_wo_no),
     +          (outbuf(33), c_qty), (outbuf(37), c_date_comp)

C  Start communications with HP Sockets

      do  10 i=1,5,1
      ilist(i)=0
      olist(i)=0
   10 continue

      flags = 0
```

```
      srcprocln = "ctl01fp"

C  NULL terminate the process logical name like a C string

      srcprocln(6:6) = char(0)

      rc = spinit(flags, srcprocln, ilist, olist)

      if ( rc .NE. 0) then
         write(*,*) "ERROR: spinit return = ", rc
         goto 999
      else
         write(*,*) "spinit successful"
      endif

C  Use an SpReadQ in a loop to read messages from the
C  sch01cp program

  100 continue

      flags = 0

      ilist(SpLOWER_TAG)=0
      ilist(SpUPPER_TAG)=0
      ilist(SpBUF_LEN)=1024
      ilist(SpTIMEOUT)=0
      ilist(5)=0

      olist(1)=0
      olist(2)=0
      olist(3)=0
      olist(4)=0
      olist(5)=0

      rc = spreadq(flags, procln, buffer, ilist, olist)

      if ( rc .NE. 0) then
         write(*,*) "ERROR: spreadq fail ", rc
         goto 999
      else
         write(*,*) "spreadq successful"
      end if
```

```
C  If we get a message with tag=9999, end the loop

      if (olist(SpTAG) .EQ. 9999) then
         go to 200
      endif

C  Display the message received

      write(*,*)
      write(*,*) "Get dispatch buffer :"
      write(*,*) "part-no = ", part_no
      write(*,*) "wo-no = ",  wo_no
      write(*,*) "qty = ", qty
      write(*,*) "start_date = ", start_date
      write(*,*) "start_time = ", start_time
      write(*,*) "stop_date = ", stop_date
      write(*,*) "stop_time = ", stop_time

C  Simulate the work complete message

      write(*,*)
      write(*,*) "***The work order has been completed  ***"
      write(*,*) "***Send work order completion info to ***"
      write(*,*) "***MRP and SCHED applications         ***"
      write(*,*)

C  Use SpSendMsg to send the work order complete message to
C  the mrp02cp process using link3 for data conversion.

      c_part_no = part_no
      c_wo_no = wo_no
      c_qty = qty
      c_date_comp = "891127"

      flags = 0

      desprocln = "mrp02cp"
      desprocln(6:6) = char(0)

      linkname = "link3"
      linkname(6:6) = char(0)
```

```
            ilist(SpCLONE_ID)=0
            ilist(SpTAG)=100
            ilist(SpBUF_LEN)=38
            ilist(4)=0
            ilist(5)=0

            olist(1)=0
            olist(2)=0
            olist(3)=0
            olist(4)=0
            olist(5)=0

            rc = spsendmsg(flags, desprocln, outbuf, linkname, ilist, olist)

            if ( rc .NE. 0) then
               write(*,*) "ERROR: spsendmsg fail ", rc
               goto 999
            else
               write(*,*) "spsendmsg successful"
            end if

      C  Use SpSendMsg to send the work order complete message to
      C  the sch02cp process using link2 for data conversion.

            flags = 0

            desprocln = "sch02cp"
            desprocln(6:6) = char(0)

            linkname = "link2"
            linkname(6:6) = char(0)

            ilist(SpCLONE_ID)=0
            ilist(SpTAG)=100
            ilist(SpBUF_LEN)=38
            ilist(4)=0
            ilist(5)=0

            olist(1)=0
            olist(2)=0
            olist(3)=0
            olist(4)=0
            olist(5)=0
```

```
      rc = spsendmsg(flags, desprocln, outbuf, linkname, ilist, olist)

      if ( rc .NE. 0) then
         write(*,*) "ERROR: spsendmsg fail ", rc
         goto 999
      else
         write(*,*) "spsendmsg successful"
      end if

      go to 100

C  End communication with HP Sockets

  200 continue

      rc = spend(flags)
      if ( rc .NE. 0) then
         write(*,*) "ERROR: spend fail ", rc
         goto 999
      else
         write(*,*) "spend successful"
      end if

  999 stop
      end
```

```
$LIST ON$

(This Pascal version of the ctl01fp process is constructed similarly
to the FORTRAN version in order to make it easy to see the correspondence
between the individual items and "procedures" in the two versions.  If
you use this Pascal adaptor template, you may wish to restructure
this program to conform to good Pascal programming construction. )


PROGRAM ctl01pp (Input, Output);

(This Pascal application adaptor (process ctl01pp) is run as a
background (daemon) process.  It hangs on the SpReadQ access routine
waiting for the message sent by the scheduling process (sch02cp) that
the work order is ready to be processed.  Process ctl01pp then sends the
message that the work order has been completed to sch02cp and mrp02cp.)

LABEL 999;

$INCLUDE '/usr/include/Sp.p.h'$          (Header file containing)
                                         (type definitions for Pascal)


TYPE
        String16 = string[16];
        String10 = string[10];
        String6 = string[6];
        String4 = string[4];

        WorkOrderIn = RECORD             (The work order information )
                part_no: String16;       (coming from sch02cp)
                wo_no: String10;
                qty: Integer;
                start_date: String6;
                start_time: String4;
                stop_date: String6;
                stop_time: String4;
        END;

        DispRecord = RECORD              (Transfer character buffer to record)
              CASE b: Boolean OF
                 TRUE : (dispchar : SpMsgBufType);
                 FALSE: (disprec  : WorkOrderIn);
              END;
```

```
        WorkOrderOut = RECORD           (The work order information )
                c_part_no:  string[18]; (going to mrp02cp and sch02cp)
                c_wo_no: string[12];
                c_qty: Integer;
                c_date_comp: string[8];
        END;

        OutRecord = RECORD              (Transfer character buffer to record)
              CASE b: Boolean OF
                 TRUE : (dispchar : SpMsgBufType);
                 FALSE: (disprec  : WorkOrderOut);
        END;

VAR
        RC, RN: Integer;                (Return Codes)
        Flags: Integer;
        LinkName: SpLnType;
        Ilist, Olist: SpIntArray5;
        DesProcIn, SrcProcIn: SpLnType; (Dest. process logical names)
        Charbuf: SpMsgBufType;          (Send buffer)
        DispBuf: DispRecord;            (Variant record variable)
        OutBuf: OutRecord;              (Variant record variable)
        ProcIn: SpLnType;               (Receive process logical name)

        Count: Integer;

$INCLUDE '/usr/include/Sp.p2.h'$        (Header file containing)
                                        (type definitions for Pascal)


BEGIN (Main Program)


(Start communications with HP Sockets by using the SpInit access
routine to initiate the ctlOIpp process. )

    For Count := 1 To 5 DO
      BEGIN
        Ilist[Count] := 0;
        Olist[Count] := 0;
      END;
```

```
        SrcProcIn := 'ctlOlpp';
        SrcProcIn[6] := chr(0);                    (Zero pad for C compatibility)
        Flags := SpDELETE_ALL;

        RC := SpInit(Flags, SrcProcIn, Ilist, Olist);

        IF (RC = 0) THEN
        BEGIN
            writeln;
            writeln('SpInit successful');
            writeln;
        END
        ELSE
        BEGIN
            writeln;
            writeln('ERROR: SpInit returns:', RC:5);
            writeln;
        END;


    (Use the SpReadQ access routine in a WHILE loop to hang on a
    message from the sch02cp process. )

        WHILE (true) DO
        BEGIN
          For Count := 1 To 5 DO
          BEGIN
            Ilist[Count] := 0;
            Olist[Count] := 0;
          END;

          Ilist[SpLOWER_TAG] := 0;
          Ilist[SpBUF_LEN] := 1024;                (The message buffer size)
          Ilist[SpTIMEOUT] := 0;                   (Number of seconds to timeout)
          Flags := 0;

          RC := SpReadQ(Flags, ProcIn, CharBuf, IList, Olist);

          IF RC = 0 THEN
          BEGIN
            writeln;
            writeln('SpReadQ successful.');
            writeln;
```

```
        END
        ELSE
          writeln('ERROR Number:', RC:5);

        if (Olist[SpTAG] = 9999) THEN
          goto 999;

        DispBuf.dispchar := CharBuf;      (Move the message buffer information)
                                          (into the record variable)

        writeln;
        writeln(' Part Number:',  DispBuf.disprec.part_no);
        writeln('Order Number:',  DispBuf.disprec.wo_no);
        writeln('    Quantity:',  DispBuf.disprec.qty);
        writeln('  Start Date:',  DispBuf.disprec.start_date);
        writeln('  Start Time:',  DispBuf.disprec.start_time);
        writeln('   Stop Date:',  DispBuf.disprec.stop_date);
        writeln('   Stop Time:',  DispBuf.disprec.stop_time);
        writeln;
        writeln('***The work order has been completed***');
        writeln;
        writeln('Send work order completion information');
        writeln('to processes mrp02cp and sch02cp');

    (Use the SpSendMsg access routine to send a message to the
    sch02cp and mrp02cp processes.)

        OutBuf.disprec.c_part_no := DispBuf.disprec.part_no;
        OutBuf.disprec.c_wo_no := DispBuf.disprec.wo_no;
        OutBuf.disprec.c_qty := DispBuf.disprec.qty;
        OutBuf.disprec.c_date_comp := DispBuf.disprec.stop_date;


        For Count := 1 To 5 DO
        BEGIN
          Ilist[Count] := 0;
          Olist[Count] := 0;
        END;

        Desprocln := 'mrp02cp';
        DesProcln[6] := chr(0);                (Zero pad for C compatibility)
```

```
LinkName := 'link3';
LinkName[6] := chr(0);

Flags := 0;
Ilist[SpTAG] := 100;                    (Tag value)
Ilist[SpBUF_LEN] := 38;                 (Length of message in bytes)

Charbuf := OutBuf.dispchar;

RC := SpSendMsg(Flags, DesProcIn, Charbuf, LinkName, Ilist, Olist);

IF (RC = 0) THEN
BEGIN
  writeln;
  writeln('SpSendMsg successful');
END;

IF RC <> 0 THEN
BEGIN
  writeln;
  writeln('ERROR Number:', RC:5);
  writeln
END;
IF RC = 0 THEN
BEGIN
  writeln;
  writeln('Message sent to process mrp02cp');
  writeln
END;


DesprocIn := 'sch02';
DesProcIn[6] := chr(0);                 (Zero pad for C compatibility)

LinkName := 'link2';
LinkName[6] := chr(0);

Flags := 0;
Ilist[SpTAG] := 100;                    (Tag value)
Ilist[SpBUF_LEN] := 38;                 (Length of message in bytes)

Charbuf := OutBuf.dispchar;
```

```
        RC := SpSendMsg(Flags, DesProcIn, Charbuf, LinkName, Ilist, Olist);

        IF (RC = 0) THEN
        BEGIN
          writeln;
          writeln('Message sent to process sch02cp');
          writeln
        END
        ELSE
        BEGIN
          writeln;
          writeln('ERROR: SpSendMsg returns:', RC:5);
          writeln
        END;


    END;

{Use the SpEnd access routine to end communication with HP Sockets. }

999:  Flags := 0;
      RC := SpEnd(Flags);
      IF (RC <> 0) THEN
      BEGIN
        writeln;
        writeln('ERROR: SpEnd returns:', RC:5);
        writeln
      END
      ELSE
      BEGIN
        writeln;
        writeln('  SpEnd was successful...');
        writeln
      END;

END. (Main Program)
```

mrp02cp

```
/*****************************************************************/
/*                                                               */
/*   Source:     mrp02cp.c                                       */
/*  $Revision:   1.1 $                                           */
/*                                                               */
/*  This is a skeleton application adaptor that initiates the    */
/*  background process mrp02cp that hangs on an SpReadQ call    */
/*  waiting for a data message from process ctl01fp.  Mrp02cp    */
/*  uses these calls: SpInit, SpReadQ, and SpEnd.                */
/*  another SpReadQ call.                                        */
/*                                                               */
/*****************************************************************/


/* ------------------------------------------------------------
 This is the maximum buffer size HP Sockets allows.
 Change it to the size that the application needs.
 ------------------------------------------------------------ */

#define MAXBUFSIZE 1024
#define MAXMSGSIZE 500

/* Global include files  */

#include <stdio.h>
#include <Sp.c.h>        /* HP Sockets header include file for C programs    */
#include "ex01.h"        /* Data and file definitions for work order management
                                examples                                     */

/* ===== Main program ===== */

main()
{
/* ------------------------------------------------------------------------
 These are suggested parameter definitions for all access routine calls.
 ------------------------------------------------------------------------ */

 int             rc;                          /* Access routine return code   */
 int             buflen;                      /* Data buffer length           */
```

```
        unsigned char   buffer[MAXBUFSIZE];        /* Data buffer received          */
        char            procln[16];                /* Process name initiate         */
        char            srcprocln[16];             /* Source process name           */
        char            desprocln[16];             /* Destination process name      */
        char            linkname[16];              /* Destination process name      */

        char            runstring[128];            /* Runstring for SpStartProcess */
        char            execinfo[256];             /* Execution information for
                                                          SpStartProcess           */

        int             flags;                     /* Flag parameter                */
        int             func;                      /* Set function for SpControl    */
        char            errmsg[MAXMSGSIZE];        /* Return error buffer           */
        int             errmsglen;                 /* Set error message length      */
        int             in_list[5],                /* Additional input parameters   */
                        out_list[5];               /* Additional output parameters */

      /* ======These are special parameters used in  this program. =======*/
        int             i;
        wo_complete_rec wo_complete_buf;


        /* =========  Start communications with HP Sockets ========= */

        flags = 0;
        strcpy(srcprocln, "mrp02cp");

        if ((rc = SpInit(flags, srcprocln, in_list, out_list)) != 0)
        {
           printf("ERROR: SpInit of %s returns %d\n", srcprocln , rc);
           exit(1);
        }
        else
           printf("  SpInit was successful...\n", rc);

      /* ================================================
        Loop forever, waiting for messages.  Stop at the special
        message with tag=9999.  Set timeout for SpReadQ to
        an infinite wait.  Default is 60 seconds.
      ================================================ */

        for (;;)
        {
```

```
        for (i = 0; i < 5; i++)
          {
              in_list[i] = 0;
              out_list[i] = 0;
          }
        flags = 0;
        in_list[SpLOWER_TAG] = 0;
        in_list[SpBUF_LEN] = MAXBUFSIZE;
        in_list[SpTIMEOUT] = 0;

        if ((rc = SpReadQ(flags, procIn, &wo_complete_buf, in_list, out_list)) >0)
        {
           printf("ERROR: SpReadQ of returns %d\n", rc);
        }
        else
        {
           printf("SpReadQ successful...\n");

      /*  ------------------------------------------------------
        When this program receives a message with the TAG
       value = 9999, it breaks out of the FOR loop.
      ------------------------------------------------------ */

           if (out_list[SpTAG] == 9999)
          break;

       printf( "From Control System, wo = %s has been completed\n",
              wo_complete_buf.wo_no);
       printf(" part-no = %s, \n qty in ascii = %s\n", wo_complete_buf.part_no,
              wo_complete_buf.qty_ascii);
          }
       }

       /* ---------- End communications with HP Sockets ---------- */

       if ((rc = SpEnd(flags)) != 0)
          printf("ERROR: SpEnd returns %d\n", rc);
       else
          printf("  SpEnd was successful...\n");

      }
```

ex01.h

```
/*****************************************************************************/
/*                                                                           */
/*  This file defines the file directories and structure for the             */
/*  example adaptors.                                                        */
/*                                                                           */
/*****************************************************************************/

/* Define the file directories. */

define MRPWOFILE "/usr/contrib/Sp/ref/programs/swofile"
define SCHWOFILE "/usr/contrib/Sp/ref/programs/dwofile"
define  SCHDISPF "/usr/contrib/Sp/ref/programs/dispatch"
define LOGINFO  "/user:password"

/* Define the structure of the work order record */
/* used in the datagen.c and sch01cp.c programs. */
/* Datagen solicits the user for this information*/
/* which the mrp01 process sends to sch01.       */

typedef struct
{
 char    wo_no[11];                    /* Work order number           */
 int     qty;                          /* Number of parts ordered     */
 char    due_date[7];                  /* Date order is due           */
 char    part_no[17];                  /* Part number                 */
} wo_rec;

/* Define the structure of the dispatch message record */
/* used in the sch01cp.c and sch02cp.c programs.       */

typedef struct
{
 char    part_no[17];
 int     qty;
 char    start_date[7];
 char    start_time[5];
 char    stop_date[7];
 char    stop_time[5];
 char    wo_no[11];
```

```
} disp_rec;

/* Define the structure of the work order complete record */
/* used in the mrp02cp.c program.                          */

typedef struct
{
 char    wo_no[11];
 char    part_no[17];
 char    qty_ascii[18];
 char    date_complete[7];
} wo_complete_rec;

/* Define the structure of the work order complete record */
/* used in the sch02cp.c program.                          */

typedef struct
{
 char    part_no[18];
 char    wo_no[12];
 int     qty;
 char    date_complete[8];
} wo_compl_s_rec;
```

datagen

```
/*****************************************************************/
/*                                                               */
/*   $Source:   datagen.c                                        */
/*   $Revision: 1.1 $                                            */
/*                                                               */
/*****************************************************************/

/* Global include   */

#include <stdio.h>
#include "ex01.h"

main()
{
   FILE *wo_fp;
   wo_rec  *wo_ptr, wo_buf;
   int i;
   char in_buf[80];


  wo_ptr = &wo_buf;

  if( (wo_fp = fopen (MRPWOFILE, "w")) == NULL)
  {
     printf("Could not open work order file %s.\n", MRPWOFILE);
     return(0);
  }

  for(;;)
  {
     printf("Do you want to add a new record into the work order file?\n");
     gets(in_buf);
     if (in_buf[0] != 'y')
 break;

     printf("Enter wo_no = (up to 10 char)\n");
     gets(in_buf);
     strncpy(wo_ptr->wo_no, in_buf, 10);
     wo_ptr->wo_no[11] = NULL;
```

```
            printf("Enter quantity ordered \n");
            gets(in_buf);
            wo_ptr->qty = atoi(in_buf);

            printf("Enter part_no = (up to 16 char)\n");
            gets(in_buf);
            strncpy(wo_ptr->part_no, in_buf, 16);
            wo_ptr->part_no[16] = NULL;

            strcpy(wo_ptr->due_date, "891206");
            if((i = fwrite(wo_ptr, sizeof(wo_rec), 1, wo_fp) ) != 1)
            {
        printf("fwrite error for %s.\n", MRPWOFILE);
        return(0);
            }
         }

         fclose(wo_fp);

    }
```

APPENDIX C

Data Definitions

DDL is used to define data that requires manipulations. These definitions are contained in a Data Definition (Ddef.def) configuration file.

Format

```
DATA_DEFINITION
     BEGIN
     <declarations>
     END;
```

Declarations

A declaration is a statement specifying the data type of one or more identifiers.

A data type is a collection of elements formed in the same way and treated uniformly. They determine these attributes for each identifier:

- A set of permissible values
- A set of permissible operations
- Storage and alignment requirements

The invention does not bind identifiers to any fixed alignment, representation, or language when the DDL declarations are compiled. The invention binds languages during the link definition specification. Alignment and physical representation of a structure in memory are determined at run time according to the semantics and machine architecture when the structure is used.

DDL data types can be:

--Simple

--Structured

Simple Types

Simple Data types are pre-defined types that indicate the format of the storage for a particular data object. The invention supports these predefined simple data types:

Ordinal	Boolean	Char	Real	User-defined
BYTE	BOOLEAN	CHAR	REAL	OPAQUE
BIT_FIELD	LONG_BOOL	STRING	LONG_REAL	
SHORT_INT	SHORT_BOOL			
INTEGER				
LONG_INT				

Ordinal

Ordinal types are: byte, bit_field, short_int, integer, and long_int.

Ordinal types are signed unless they are preceded with the keyword UNSIGNED. If ordinals are UNSIGNED, they can represent only positive values.

The unsigned keyword may also change the algorithms that the invention uses to calculate alignment, storage, representation and type conversion requirements. For example, a short_int on an HP9000 computer may have the values 0 ....65535.

Although the actual sizes vary according to the machines used, these are the general effects of the UNDERSIGNED keyword with type conversions:

- When the source integer's value is greater than can be physically represented by the destination type, the destination value will be set to the largest number that type can represent.

Example:

A short_int (16 bits) with the value of 346 at the source that is converted to a SIGNED byte (8 bits) will have the value of 127. If it is converted to an UNSIGNED byte, it will have the value of 255.

- When the source integer's value is smaller than can be physically represented by the destination type, the destination integer will be set to the smallest number that type can represent.

Example:

A short int (16 bits) with the value of -346 at the source that is converted to a signed byte (8 bits) will have the value of -128. If it is converted to an unsigned byte, its value will be 0.

BYTE

A byte type represents individual characters of data as integers. This type is a convenient method for denoting very small integers. It represents the smallest individually addressable memory element in 8-bit byte machine architectures. The subrange of integers for the byte type on the HP9000 computers is -128...127.

BIT_FIELD[1-32]

A bit_field type denotes an integer that is represented by an arbitrary number of bits. Bit fields can extend from 1 through 32 bits. Bit fields can be signed or unsigned. The size of an integer on a given architecture is the largest bit field allowed. Bit fields are placed in storage locations from the most significant to the least significant bits.

Bit fields may not cross or straddle a word boundary as it is defined for a particular machine architecture. If a declaration causes a bit field to exceed the current word, the bit field is put into the beginning of the next word and the remaining bits in the current word are skipped.

SHORT_INT
INTEGER
LONG_INT

Short_int, integer, and long_int types are simple types whose ranges are determined by subranges of the negative and positive integers. The actual subranges are determined by the machine architecture on which they are referenced. Like other ordinal types, you can modify the subrange with the keyword UNSIGNED. On the machine architectures supporting these data types, the short_int has the smallest width of subranges, and the long_int has the largest. For example, these data types have these subranges of integers on the HP9000 computers:

SHORT_INT	-32,768 . . . 32,767
INTEGER	-2,147,483,648 . . . 2,147,483,647
LONG_INT	-2,147,483,648 . . . 2,147,483,647

In C on the S300 and S800, integer is the same length as long_int.

Note that where an architecture or language does not possess one or more of these data types, the nearest equivalent ordinal type is used instead.

Boolean

SHORT_BOOL
BOOLEAN
LONG_BOOL

Short_bool, boolean, and long_bool are simple types that indicate logical values. These data types are equivalent to the logical data type in FORTRAN. The two possible values for these types are TRUE and FALSE. The representation and alignment of a boolean value depends on the language and machine on which it is used.

Char

CHAR

A char type represents individual characters in the 8-bit ASCII character set. A character literal is either a single character enclosed by single quotation marks or an escape character. All character types are assumed to contain ASCII data. Use the byte type if a structure contains binary data. The particular character set representation is determined by the machine architecture where this data is produced or consumed.

STRING

A string type represents a sequence of characters. The way the character sequence is represented depends on the language that produces or consumes the data. Strings have an upper limit of 1024 characters.

The string type consists of the keyword STRING and an integer containing the maximum length of the string enclosed in square brackets. For C, the maximum length must include one extra character for the terminating null character. For Pascal and FORTRAN, the maximum length is the same as would be defined in a Pascal or FORTRAN program.

In C language, the string type is a sequence of characters terminated by a NULL where NULL has a value of zero. The NULL terminator is considered part of the string. In FORTRAN, a string type is a sequence of characters that is blank-padded to the string's maximum length. In Pascal, the string type is a sequence of characters preceded by an ordinal specifying the current length of the string.

Note that the Pascal language on HP9000/S300 and S800 has an explicit string data type separate from a packed array of character data type. These data types are not the same thing to DDL. Use the DDL string type for the Pascal string type. Use the DDL packed array of character declaration for Pascal packed array of character data elements.

Real

REAL
LONG_REAL

Real and long_real types represent a subset of the real numbers. A real type is generally smaller than a long_real. The range (precision) for this data type depends on the machine architecture on which the data is used.

User-defined

OPAQUE An opaque type denotes binary data which the invention should not alter The opaque type is always represented by eight binary bits. The purpose of the opaque type is to allow unaltered transfer of binary data between different systems. Translating binary data (such as Programmable Logic Controller ladder logic programs) to suit a particular machine architecture would corrupt the data and prevent its being downloaded to its originator.

Example of Simple Data Type

This is an example of a data definition using the simple data types BYTE and LONG_BOOL:

```
DATA_DEFINTION
BEGIN
     T1 = BYTE;
     Stuff = LONG_BOOL;
END;
```

Structured Types

Structured types are built from simple data types or other structured types. DDL supports two structured types:

- ARRAY
- RECORD

Pascal structured types may be specified as packed. The PACKED specification tells the DDL compiler that the storage requirements for the components of the given array or record have been or should be optimized. The particular storage layout and alignment algorithms depend on the language and machine architecture that produces or consumes the array or record. The PACKED specification has no effect on C or FORTRAN elements.

ARRAY

An array consists of components that are all the same type. You can use other structured types to build structures such as multi-dimensional arrays and arrays of records.

Format

identifier = [PACKED] ARRAY [<dimensions>] OF <type>

where:

type — Simple types, arrays, records, or other previously defined DDL types

The dimension specification indicates the maximum number of elements for a given dimension of the array. DDL always assumes the lower bound of one for each dimension of an array. The dimension length may be any integer constant. You can access each element of an array with the index of the array element.

An array is said to be multi-dimensional if its definition contains more than one dimension. The invention defers its assumptions about th order of the array element until a specific language is bound to the source storage and destination data structures during a data manipulation specification. The specific language determines how the multi-dimensioned arrays are stored in memory (row-major or column-major). FORTRAN is column-major. C and Pascal are row-major.

RECORD

A record is a sequence of named members that are not necessarily of the same type. Each member is called a field of the record and has its own identifier. Fields in different records can have identical names but represent different objects. You access a field in a record with the appropriate field selector (".").

Fields are placed in physical storage in their order of declaration in the record definition. A field's offset is the distance from the start of the record to the beginning of the field.

Format:

identifier = [PACKED] RECORD OF [<fields>]

A field definition consists of an identifier, a colon, and a type. Recursion in any form is not allowed. You may not declare a record element which has itself as the type.

Example

This example shows records with members of simple and structured types.

```
name_info = RECORD OF [
        first   :       STRING[30];
        middle  :       STRING[30];
        last    :       STRING[30];
        ];

addr_info = RECORD OF [
        street  :       STRING[80];
        city    :       STRING[30];
        state   :       STRING[16];
        zip     :       INTEGER;
        ];

person_info = RECORD OF [
        name    :       name_info;
        addr    :       addr_info;
        married :       BOOLEAN;
        numkids :       BYTE;
        kidnames:       ARRAY[100] OF name_info;
        ];
```

Language Dependencies

Table C1/ C Data Types and DDL

HP C Type	DDL Type	Comments
char (ASCII)	char	The DDL char data type is always assumed to contain ASCII data, not binary (integer) values.
char (signed data)	byte	The DDL byte type is always assumed to contain an 8-bit binary (integer) value and may be signed or unsigned.
unsigned char	unsigned byte	
short	short_int	
unsigned short	unsigned short_int	
int	integer	
unsigned int	unsigned integer	
long	long_int	
unsigned long	unsigned long_int	
float	real	
double	long_real	
char array	string[n]	For transformation purposes, the DDL string type is always assumed to contain ASCII data.
array	array	For the S300 and the S800, arrays are aligned by element type.
struct	record	These are the alignment rules for records: S300 - nested record - 2 bytes S300 - unnested record - 4 bytes S800 - largest field
char (as boolean)	short_bool	Boolean types contain one of only two values: TRUE or FALSE. When a DDL boolean type is used, the invention applies the C language-imposed rules for determining if the value is true or false.
int (as boolean)	long_bool	
short (as boolean)	boolean	
long (as boolean)	long_bool	
enum	integer	

(Table continues on next page)

(Continued from previous page)

HP C Type	DDL Type	Comments
bit fields	bit_field	C supports a bit field data type. DDL does not explicitly support this data type at present. DDL does support an idealized bit field type with the following assumptions: Within a record, a bit_field's size remains what it is; it is not assummed to be rounded up to any byte boundary. Its alignment is considered to be 1 bit. Within an array, a bit_field's size is assumed to be rounded up to the nearest 8, 16, or 32 bits. Its alignment is assumed to match this rounded-up size.

There is no language counterpart for the DDL opaque data type. The invention will not perform transformations on this type. Its representation is always an unsigned 8-bit octet. An opaque data type is assumed to begin at the next byte boundary.

Table C2 shows the DDL equivalents for FORTRAN 77 data types.

Table 5-2. FORTRAN 77 Data Types and DDL

FORTRAN 77 Type	DDL Type	Comments
Logical*2	boolean	
Logical*4	long_bool	
Integer*2	short_int	
Integer*4	integer	
Integer*4	long_int	
Real*4	real	
Real*8	long_real	
Character*N	string[n]	For transformation purposes, the DDL string type is always assumed to contain ASCII data.
Array	array	For the S300 and S800, arrays are aligned by element type.
Common Blocks Equivalenced data	record	These are the alignment rules for records: S300 - first field S800 - first field

Table C-3 shows the DDL equivalents for Pascal data types.

Table C-3 Pascal Data Types and DDL

Pascal Type	DDL Type	Comments
Boolean	short_bool	Boolean types contain one of only two values: TRUE or FALSE. When a DDL boolean type is used, the invention applies the Pascal language-imposed rules for determining if the value is true or false.
Boolean	boolean	
Boolean	long_bool	
-128..127	byte	The DDL byte type is always assumed to contain an 8-bit binary (integer) value and may be signed or unsigned.
Shortint	short_int	There is no Pascal shortint type on the S300. S300 int subrange is -32768...32767.
Integer	integer	
Integer	long_int	There is no Pascal long_int type on the S800.
Char	char	The char type is always assumed to contain ASCII data, not binary (integer) values.
int subrange	bit_field	Bit fields may be from 1 to 32 bits in length. The invention uses Pascal's rules for aligning and sizing bit fields. Sizing and alignment rules for bits fields are shown below. Independent bit fields or those contained in an unpacked record or array: S300 - Size is rounded up to the nearest 16 or 32 bits; alignment is the same as the resulting size. S800 - Size is rounded up to the nearest 8, 16, or 32 bits; alignment is the same as the resulting size. Bit fields that are elements in a packed array: S300 - Size and alignment are rounded to the nearest 1, 2, 4, 8, or 16 bits. If the bit field is greater than 16 bits in size, alignment is 16 bits and size is rounded up to 32 bits. S800 - Size and alignment are rounded up to the nearest 1, 2, 4, 8, 16, or 32 bits. Bit fields that are elements in a packed record: S300 - Size is the actual size of the bit field and alignment is 1 bit. S800 - Size is the actual size of the bit field and alignment is 1 bit.
int subrange	unsigned bit_field	
Real	real	
Longreal	long_real	

(Table continues on next page)

(Continued from previous page)

Pascal Type	DDL Type	Comments
String(N)	string[n]	For transformation purposes, the string type is always assumed to contain ASCII data
Array	array	These are the alignment rules for arrays: S300 - nested array - 2 bytes S300 - unnested array - 2 or 4 bytes (2 if the element size is 2 bytes or less) S800 - arrays are aligned by element type
Record	record	These are the alignment rules for records: S300 - nested record and packed - 2 S300 - nested record and unpacked - 1 or 2 (1 if record size is 1 byte or less) S300 - unnested record and packed - 2 or 4 (2 if record size is 2 bytes or less) S300 - unnested record and unpacked - 1, 2, or 4 (1 if record size is 1 byte or less, 2 if record size is 2 or less) S800 - largest field
SETS	-	DDL does not support enumerated types or sets. However, since Pascal represents these types as an ordinal type, you can use DDL to define a reasonable facsimile.

Example

This is an example of a string as defined in DDL, C, FORTRAN, and Pascal.

DDL

```
Rec1 = RECORD OF [
        b1 : BYTE;
        s1 : SHORT_INT;
        a3 : ARRAY[3] OF CHAR;
        i1 : INTEGER;
        ];
```

C

```
struct {
        char b1;
        short s1;
        char a1[3];
        int i1;
} rec1;
```

FORTRAN

```
C  Elements of structure

   CHARACTER*1 b1
   CHARACTER*3 a1
   INTEGER*2 s1
   INTEGER*4 i1

C  Use this to be sure that the first element of the
C   structure is on a 4-byte alignment

   INTEGER*4 d1

C  Call SAR with this buffer

   CHARACTER*30 cb1

   EQUIVALENCE (d1, b1, cb1)
   COMMON /rec1/ b, s1, a1, i1
```

Pascal

```
trec1a = RECORD
      b1 : char;
      s1 : shortint;
      a1 : packed array[1...3] of char;
      i1 : integer;
      END;

buff : array[1..200] of char;

{ call SAR using buf }

trec1 = RECORD CASE INTEGER OF
      1 : (r1 : trec1a);
      2 : (buf : buff);
      END;
```

Data Definition Declarations

```
<data-def-section>   ::= DATA_DEFINITION BEGIN <declarations> END ;

<declarations>       ::= <declaration>
                     |   <declarations> <declaration>

<declaration>        ::= <identifier> = <type> ;

<type>               ::= [PACKED] <structured>
                     |   <simple-type>
                     |   [PACKED] <identifier>

<simple-type>        ::= [UNSIGNED] <ordinals>
                     |   <non-ordinals>

<ordinals>           ::= SHORT_INT
                     |   INTEGER
                     |   LONG_INT
                     |   BYTE
                     |   BIT_FIELD [ <bit-length> ]

<non-ordinals>       ::= SHORT_BOOL
                     |   BOOLEAN
                     |   LONG_BOOL
                     |   CHAR
                     |   REAL
                     |   OPAQUE
                     |   LONG_REAL
                     |   STRING [ <length> ]

<structured>         ::= ARRAY [ <dimensions> ] OF <type>
                     |   RECORD OF [ <fields> ]

<fields>             ::= <field> ;
                     |   <fields> <field> ;

<field>              ::= <identifier> : <type>

<dimensions>         ::= <length>
                     |   <dimensions> , <length>

<length>             ::= <decimal-const>

<bit-length>         ::= 1 | 2 | ... | 32
```

APPENDIX D

Data Manipulations

After the data structures to be passed between applications in DDL, are defined, the Data Manipulation Language should be used to define the manipulation, the invention must perform on the structures. These manipulations are defined in the Data Manipulation Definition configuration file (Dman.def). They reference the data definitions defined in the Data Definition configuration file (Ddef.def).

DML has two declaration parts:

- The header section

- The manipulation statements section

FORMAT

```
DEFINE_MANIPULATION <identifier>;
SOURCE_DATA: <identifier>
DESTINATION_data: <IDENTIFIER>:

BEGIN MANIPULATION
<manipulation statements>;
END_MANIPULATION;
```

Header Section

The header section defines:

- The name of the manipulation

- The name of the DDL definition of the source data

- The name of the DDL definition of the destination data

Manipulation Statements Section

The manipulation statement section contains the instructions for translating from one data structure format to another. The instructions or statements are executed sequentially in their order in this section.

There are two kinds of statements in the declaration section:

-Assignment statements

-Move statements

Assignment Statement

Assignment statements assign literal values and operators for accessing arrays and structures.

Format

```
structure = value;
```

You use assignment statements primarily to set a default value (constant or literal) for a destination field.

The literal value assigned to the destination field may reference only the destination DDL data type which must be a simple type or an array of simple type. If the destination field is an unqualified array, the literal value specified sets all the elements of the array.

The literal value must be appropriate for the destination field type. For example, a destination field that is an integer type must be assigned a literal that is an integer. The literal value must also physically fit within the legal range of values for the destination type. For example, a signed byte as the destination type must be assigned a value between -128 and 127. Any other value is illegal.

Constants

Constants are primary expressions whose literal or symbolic values do not change. Each constant has a value and a type determined from its form. The invention evaluates constants at compile time.

Constants can have these values: floating point, integer, character, and string literal.

Floating-Point

Floating-point constants represent floating-point values. They consist of a coefficient and an optional scale factor (exponent). The coefficient is a decimal point with at least one digit preceding or following it. The optional exponent is introduced by either the E (or e) character or the L (or l) character (for an extended floating point number) followed by the scale factor. Maximum and minimum values and significant digits depend on the particular machine architecture used.

These are examples of floating-point constants:

3.14159265358	.31459e+1	7.E5
.876	2.	

Integer

Integer constants represent integer values. This constant consists of a sequence of digits with an optional preceding sign character.

Use the L suffix to force a constant to a long integer (example: 37L or 37l). Begin hexadecimal constants with 0x (the x character case does not affect its value). You can force hexadecimal constants to long integers with the L suffix. Examples of hexadecimal constants are:

0xAAF OxbAd OX1AF7L

Character Character constants are any characters except the single quote ('), the double quote ("), and the backslash (\). Use the backslash character (\) followed by up to three octal digits for octal escape sequences. Note that this octal representation does not require a leading zero and you can use fewer than three digits. The two character constants below are identical; each represents the decimal value of 68:

'D' and '\104'

Use the backslash character followed by an x and up to two hexadecimal digits for hexadecimal escape sequences. The two character constants below are identical:

'C' and '\x43'

In both octal and hexadecimal representations, the backslash and following digits are converted into a single 8-bit character which is stored in the character constant.

The escape sequence representation of new line, single quote, backslash, and double quote special characters are \n, \', \\, and \" respectively.

String Literal A string literal constant is a sequence of zero or more characters enclosed in double quotation marks (" and ").

Use the escape character sequence to represent the double quote character appearing in a string literal as in the example below.

" Samantha said \"Shorten the sentence\" "

Consecutive double quotation marks represent null strings.

Note that you cannot escape the escape character itself. You must represent it as a hexadecimal or octal constant (example: \x5c)

The invention determines the type and representation of the string by the context in which it is used and stores it correctly for its given language and machine architecture.

Move Statements

Move statements let you move a source structure into a destination structure with a transparent correction for language and machine architecture incompatibilities. The move statement performs these tasks:

- Corrects for differences in the languages of the source and destination structures
- Corrects for differences between the machine architectures that produce the source and consume the destination structures
- Provides type conversions between the structure components where necessary.

Format

MOVE <source> TO <dest> [USING "<fmt-descr>"]

The move statement consists of the keyword MOVE, the data source structure, the keyword TO, and the data destination structure. Use the optional keyword USING followed by a format descriptor to convert between numeric and ASCII structure components.

Source and dest refers to the name of any structured type or any element (field) of a structured type. The source and destination structures are defined with DDL in the Ddef.def file and specified in the header section of the data manipulation definition. Note that destination to destination moves and source to source moves are not allowed.

Array Subscripting

All elements of the array will be moved if the array is not subscripted. You can make a subscripted reference to a single element in an array.

DML uses the language declared in Link.def to determine the storage method of the individual elements of an array. Both C and Pascal store their array elements in row-major order. This is the opposite of FORTRAN which stores array elements in column-major order. The Link Definition file (Link.def) lets you select C, Pascal, or FORTRAN as the language that determines the storage of multi-dimensional arrays in the source and destination DDL data definitions.

Structure Members

You can move the entire structure by specifying an unqualified structure name. You can move a subset of the structure by using the field selector operator (.) to qualify the structure.

This is the convention that the move statement uses to determine which source structure components are moved into which destination structure components: it converts any component (field) in the source structure

that has a name identical to a destination structure component (field) to the appropriate destination component type and then moves it. This definition can be applied in a nested fashion, that is, the components of structured components also follow the same rules.

The move statement ignores component names existing in the source but not the destination structure. The move statement initializes component names existing in the destination but not the source structure to default values appropriate for the type.

Example

These are DDL definitions for source and destination data structures.

```
name_info1 = RECORD OF [
                first   : string[15];
                last    : string[30];
                middle  : string[15];
                ssn     : array[9] OF char;
                ];

name_info2 = RECORD OF [
                last    : string[30];
                middle  : char;
                first   : string[20];
                ssn     : long_int;
                ];

misc_struct = RECORD OF [
                rdata   : real;
                idata   : integer;
                ];

source = RECORD OF [
                rec_size        : short_int;
                descript        : STRING[30];
                rec_data        : array[10,5] OF integer;
                com_struct      : misc_struct;
                different1      : name_info1;
                ];

destn = RECORD OF [
                rec_type        : char;
                rec_size        : long_int;
                rec_data        : array[10,8] OF STRING[20];
                com_struct      : misc_struct;
                different2      : name_info2;
                ];
```

These are examples of DML move statements moving the source data to the destination data defined above.

```
MOVE source TO destn;
MOVE source.different1 TO destn.different2;
MOVE source.com_struct.rdata TO destn.com_struct.idata;
MOVE source.descript TO destn.rec_data[1,8];
```

Figure D1 shows what happens when you move source to destn.

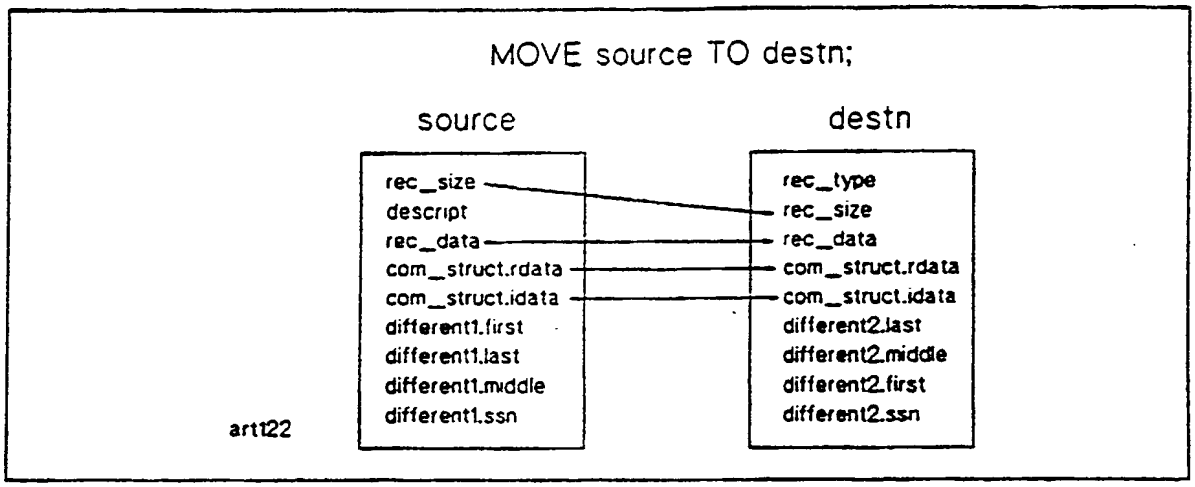


Figure D1 **Move Statement Example 1**

Fig.D2 shows what happens when you move the field source.different1 to the destination field destn.different2.

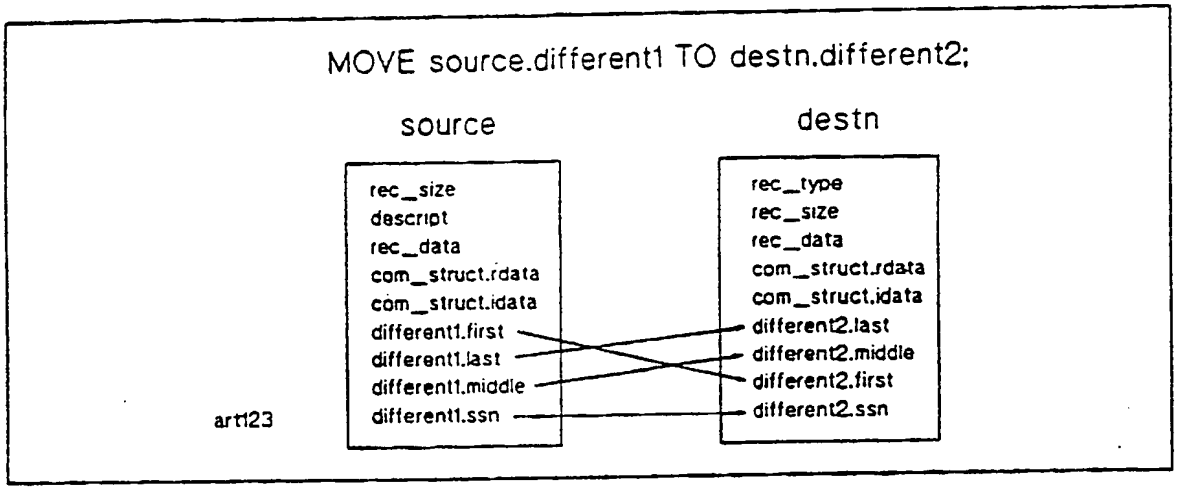


Fig. D2 Move Statement Example 2

Fig.D3 shows what happens when you move the com_struct.rdata field in the source structure to the com_struct.idata field in the destination structure.

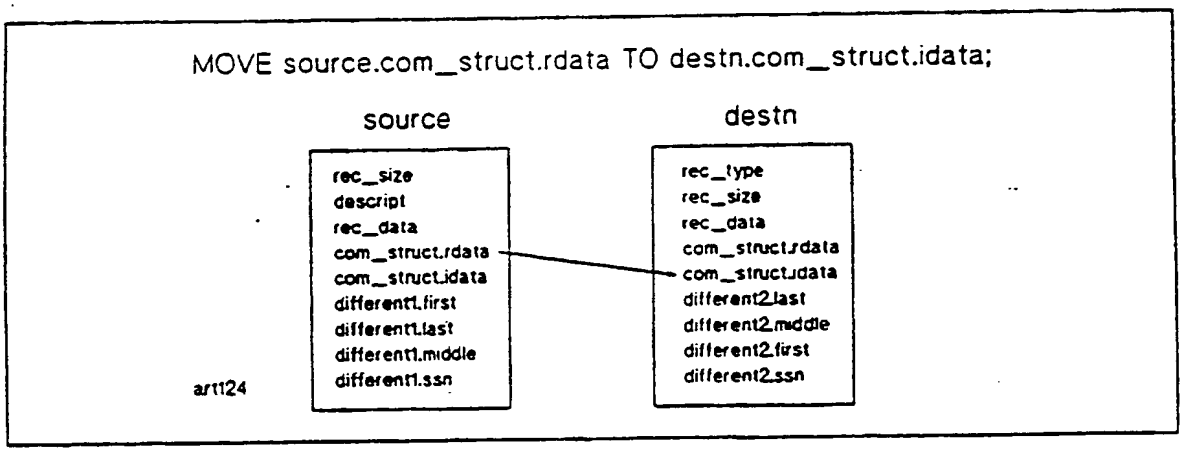


Fig. D3 Move Statement Example 3

DML Type Conversions

. For example, the move statement handles type conversions between fields with different numeric types and between ASCII and numeric types.

Refer to the following tables for the rules for these data type conversions:

- Boolean (boolean to boolean, ordinal to boolean)
- Numeric to numeric
- Floating point (integer to floating point and floating point to integer)
- ASCII to numeric and numeric to ASCII

Boolean Type Conversions

Table D-4 shows the action for boolean type conversions.

Table D-4 Boolean Type Conversions

Source Field Type	Destination Field Type	Action
Short boolean Boolean Long boolean	Boolean	If the source field type is a boolean, the destination field type must be a boolean.
Integer	Boolean	The source field type language determines whether the integer type is true or false. For example, if the source language is Pascal, a zero value means FALSE is placed into the destination field; a value of one places TRUE in the destination field. Any other value yields an unpredictable destination field value with a warning logged at the source node. If the source language is C, a value of zero is FALSE and a non-zero is TRUE.

Numeric to Numeric Type Conversions

Numeric to numeric type conversions involve handling the differences in the sizes and the signs of the source and destination data fields.

Table D-5 shows the rules used in converting from one numeric type to another numeric type.

Table D-5 Numeric to Numeric Type Conversions

Numeric Type	Source/Destination Size Difference	HP Sockets Action
Integer	The value of the source is larger than the destination can hold	HP Sockets sets the destination to the maximum value the destination can hold. **Example:** If an unsigned byte holding 250 is moved to a (signed) byte, HP Sockets sets the value in the byte to +127.
	The value of the source is smaller than the destination can hold	HP Sockets sets the destination to the minimum value the destination can hold. **Example:** If a (signed) byte holding -46 is moved to an unsigned byte, HP Sockets sets the value in the unsigned byte to 0.
Floating point	The source's value is too large to be represented in the destination (overflow)	HP Sockets sets the destination to the maximum value the destination can hold **Example:** If the source is a long_real with a value of 6.3E+50 and it is moved to a real (on the S300), HP Sockets sets the destination to 3.402_E+38 because that long_real value cannot be represented in a (S300) real.
	The source's value is too small to be represented in the destination (underflow)	HP Sockets sets the destination to zero. **Example:** If the source is a long_real with a value of .1E-300 and it is moved to a real, HP Sockets sets the destination to 0 because .1E-300 cannot be represented in a real.

Floating Point Type Conversions

These are the general rules for conversions between numeric types (integer and floating point) :

- If the value of the source field is greater than can be represented physically by the destination, the destination is set to the maximum permissible value (infinity if supported by the architecture). A run time warning is generated and logged.
- If the value of the source is smaller than can be represented physically by the destination, the destination is set to the minimum permissible value (negative infinity if supported by the architecture). A run time warning is generated and logged.

Table D-6 demonstrates the rules for integer to floating point and floating point to integer type conversions.

Table D-6 Floating Point to Floating Point Type Conversions

Source Field Type	Destination Field Type	HP Sockets Action
Integer	Floating point	If the integer's precision is greater than a real type can represent, the least significant digits are lost
Floating point	Integer	HP Sockets truncates the fractional part

ASCII and Numeric Type Conversions

You can convert a numeric type to ASCII and the reverse if one of the fields is a string or char type and the other is an integer or floating point type. Table D-7 shows these conversions.

Table D-7 ASCII and Numeric Type Conversions

Source Field Type	Destination Field Type	Action
Numeric: Integer or floating point	ASCII: String or char array	Source value is converted to an ASCII format generic to the numeric type. If the destination field is not large enough to hold the ASCII data, asterisks (*) are put into the field.
Numeric: Integer or floating point	ASCII: Multi-dimensioned array of char	The number's ASCII representation is placed into each row or column of the destination array according the destination language. That is, the array is filled by rows if the destination language is C or Pascal, and filled by columns if it is FORTRAN. If the source field is an array, each element of the source array is 1) processed in row-major order for C and Pascal and column-major order for FORTRAN, 2) type converted to ASCII, and 3) its ASCII representation placed in the appropriate row or column of the destination array.
ASCII: String or char array (Only decimal, octal, hexadecimal, or sign characters allowed. For floating point E, e, and dot (.) allowed.)	Numeric: Integer or floating point	Source value is converted to numeric type. If the destination field cannot contain the numeric value, the max or min values are used depending on the value of the source field.
ASCII: Multi-dimensioned array of char (Only decimal, octal, hexadecimal, or sign characters allowed. For floating point E, e, and dot (.) allowed.)	Numeric: Integer or floating point	The number represented in ASCII in the source is placed into each row or column of the destination array according to the destination language. That is, the array is filled by rows if the destination language is C or Pascal, and filled by columns if it is FORTRAN. If the source field is an array, each element of the source array is 1) processed in row-major order for C and Pascal and column-major order for FORTRAN, 2) type converted to numeric, and 3) the number is placed in the appropriate row or column of the destination array.
ASCII	Numeric array	Each element of the array is set to the ASCII value.

Example

This is an example of numeric to ASCII type conversions. This example shows moving the date consisting of three integers into an ASCII string and an ASCII array of char.

This structure in a C program contains information about an operator that includes the date employment started:

```
struct {
      char Operator[20];
      int Month;
      int Day;
      int Year;
} id, rid;
```

The manipulation that we want to do is to move the operator and date information into both of these 2 structs:

```
struct {
      char Operator[15];
      char Date[11];
} r1, rr1;

struct {
      char Operator[15];
      char Date[20];
} r1a, rr1a;
```

These are numeric to ASCII conversions, moving integers from structure id into a character string (in structure r1) and an array of char (structure r1a).

These are the DDL data definitions for the three structures:

```
DATA_DEFINITION
BEGIN

/* Define structure ID as a record named
 input_data in DDL */

input_data = RECORD OF [
      Operator    :     STRING[20];
      Month       :     INTEGER;
      Day         :     INTEGER;
      Year        :     INTEGER;
```

```
      ];

/* Define structure id as a record named report1
in DDL */

report1 = RECORD OF [
      Operator    :     STRING[15];
      Month       :     ARRAY[2] OF CHAR;
      Slash1      :     CHAR;
      Day         :     ARRAY[2] OF CHAR;
      Slash2      :     CHAR;
      Year        :     STRING[5];
      ];

/* Define structure r1a as a record named
report1a in DDL */

report1a = RECORD OF [
      Operator    :     ARRAY[15] OF CHAR;
      Year        :     ARRAY[4] OF CHAR;
      Dot1        :     CHAR;
      Month       :     ARRAY[2] OF CHAR;
      Dot2        :     CHAR;
      Day         :     STRING[3];
      ];
END;
```

i_to_report1 is the manipulation definition for moving the input data structure into the report1 structure. This displays the operator name as a string, and the date in American format (mm/dd/yy). It uses the assignment statement to set the characters Slash1 and Slash2 to '/'.

```
DEFINE_MANIPULATION i_to_report1;
      Source_Data : input_data;
      Destination_Data : report1;

      Begin_Manipulation

      move input_data to report1;
      report1.Slash1 = '/';
      report1.Slash2 = '/';

      End_Manipulation;
```

i_to_report1a is the manipulation definition for moving the input data structure into the report1a structure. This displays the operator's name as an array of char, and the date in European format (yy.mm.dd). It uses the assignment statement to set the characters Dot1 and Dot2 to '.'.

```
DEFINE_MANIPULATION i_to_report1a;
      Source_Data : input_data;
      Destination_Data : report1a;

      Begin_Manipulation

      move input_data to report1a;
      report1a.Dot1 = '.';
      report1a.Dot2 = '.';

      End_Manipulation;
```

If structure id is initialized to these values:

```
strcpy(id.Operator, "John Skupniewicz");
id.Month = 1;
id.Day = 30;
id.Year = 1990;
```

The converted structure will display in Report1 as:

```
Operator is John Skupniewi
Date is 1 /30/1990
```

and in Report1a as:

```
Operator is John Skupniewic
Date is 1990.1 .30
```

ASCII Format

You can control the format of ASCII values by means of a format descriptor specified with the move statement. The format descriptor controls type conversions at either the source or the destination. When the ASCII data is the source field, the format descriptor describes the way the source data is read. When the ASCII data is the destination, it describes the way it should be generated.

Format ASCII values with the keyword USING followed by a format descriptor as in this example:

```
MOVE <source> TO <dest> [USING "<fmt-descr>"]
```

Refer to the Move Statement description for explanation of the components in the move statement.

Valid formats are:

%d	Decimal integer (signed or unsigned)
%u	Unsigned decimal integer
%o	Octal integer (signed or unsigned)
%x	Hexadecimal integer (signed or unsigned)
%[m.n]f	Floating point number in the form [-]iiii.ffff
%[m.n]e	Floating point number in the form [-].ffffffe[-]xx

where:

m	Designates the field width in characters including any signs or decimal point
n	Designates the precision

m must be greater than n.

m and n are optional. If they are not specified, m has a default of 14 and n has a default of 6. If the destination field cannot hold a 14-character width, the width is decreased starting with the digits to the right of the decimal and continuing to the digits to the left of the decimal if necessary. A warning message is issued.

Example

This is an example of using a MOVE statement to convert a buffer from a real type to an ASCII type using a real number format descriptor:

```
MOVE buffer1.real TO buffer2.ascii USING "%16.2f"
```

Table D-7 describes results of other actions in formatting ASCII data.

Table D-7 Formatting ASCII Data

Action/Condition	Result
Source data does not match the format specified	No conversion performed. Warning message is generated.
Format descriptor left out of the MOVE command	Default formats used: %d for ASCII to ordinal and ordinal to ASCII %14.6f for ASCII to real or real to ASCII

Language Specifics in the Move Statement

In addition to type conversion, the move statement also resolves the differences in the way different languages represent simple and structured types. For example, while the C language represents FALSE as any zero value and TRUE as any non-zero value, HP Pascal represents FALSE as zero and TRUE as one in the rightmost bit of a byte.

When both the source and destination fields are booleans, MOVE converts data as appropriate for the language bound to the data structure in the source and destination specification section of this data manipulation definition.

MOVE corrects for any difference in the order the elements of multi-dimensional arrays are stored. It uses one of two storage methods: row-major or column-major. In column-major storage, the first dimension of the array varies most rapidly in terms of the sequence of elements stored in memory. In row-major order, the last dimension of the array varies most rapidly.

When the source is a multi-dimensional array, DML examines the language of the destination field for its storage method. If the arrays use different storage methods, the move statement converts the source array to the method used by the destination field's language. It moves the converted source array into the destination field up to the maximum number of elements in the source or the destination array, whichever is first. The source array is truncated if it is longer than the destination field. If the source is smaller than the destination, remaining bytes in the destination are not touched. Uninitialized destination array elements are set to default values.

The number and size of elements in the source and destination arrays do not need to agree for proper conversion. DML accesses the source array element by element according to the language bound to it. The destination array is filled according to the row-major or column-major rules of its language. The following example in which the source(sarray) is in FORTRAN and the destination (darray) in C, shows this conversion.

Example

At the source, sarray is stored in memory in this order:

```
sarray[1,1]    sarray[2,1]
sarray[1,2]    sarray[2,2]
sarray[1,3]    sarray[2,3]
```

At the destination, darray is stored in memory in this order:

```
darray[1,1]      darray[1,2] darray[1,3]
darray[2,1]      darray[2,2] darray[2,3]
```

You made these DDL and DML declarations:

```
sarray = ARRAY[2,3] OF BYTE;
darray = ARRAY[2,3] OF BYTE;

MOVE sarray TO darray;
```

This would be the result of the MOVE:

```
darray[1,1] = sarray[1,1];
darray[1,2] = sarray[1,2];
darray[1,3] = sarray[1,3];
darray[2,1] = sarray[2,1];
darray[2,2] = sarray[2,2];
darray[2,3] = sarray[2,3];
```

Declare both the source and destination as one-dimensional arrays to place the source array elements into a destination array while retaining the source array in its original memory order.

MOVE takes care of any alignment differences induced by language or architecture.

Data Manipulation Language Syntax

```
<manip-def-section>   ::=  DEFINE_MANIPULATION <identifier> ; <main-body>

<main-body>           ::=  <source-spec> <dest-spec> <statement-sect>

<source-spec>         ::=  SOURCE_DATA: <identifier> ;

<dest-spec>           ::=  DESTINATION_DATA: <identifier> ;

<statement-sect>      ::=  BEGIN_MANIPULATION <statements> END_MANIPULATION ;

<statements>          ::=  <statement> ;
                      |    <statements> <statement>

<statement>           ::=  <expression-statement>
                      |    <xlation-statement>

<xlation-statement>   ::=  <move-statement>

<move-statement>      ::=  MOVE <postfix-expression> TO <postfix-expression>
                               [USING " <fmt-descr> "]

<expression-statement>::=  <postfix-expression> = <constant>
                      |    <postfix-expression> = <string-literal>

<postfix-expression>  ::=  <primary-expression>
                      |    <postfix-expression> [ <primary-expression> ]
                      |    <postfix-expression> . <identifier>

<primary-expression>  ::=  <identifer>
                      |    <constant>
                      |    <string-constant>

<fmt-descr>           ::=  %d
                      |    %o
                      |    %x
                      |    %u
                      |    %[ <real-descr> ]f
                      |    %[ <real-descr> ]e

<real-descr>          ::=  <width-len> . <digit-len>

<width-len>           ::=  <decimal-const>

<digit-len>           ::=  <decimal-const>
```

Lexical Elements

<token>	::=	<*keyword*>
	|	<*identifier*>
	|	<*constant*>
	|	<*operator*>
	|	<*punctuation*>
<identifier>	::=	<*nondigit*>
	|	<*identifier*> <*nondigit*>
	|	<*identifier*> <*digit*>
<nondigit>	::=	any character from the set: _ [a..z] [A..Z]
<digit>	::=	0 | 1 | ... | 9
<constant>	::=	<*floating-pt-const*>
	|	<*integer-const*>
	|	<*character-const*>
	|	<*string-literal*>
<floating-pt-const>	::=	[<*sign*>] <*fraction-const*> [<*exp-part*>]
	|	[<*sign*>] <*digit-sequence*> <*exp-part*>
<fraction-const>	::=	[<*digit-sequence*>] . <*digit-sequence*>
	|	<*digit-sequence*> .
<exp-part>	::=	E [<*sign*>] <*digit-sequence*>
	|	L [<*sign*>] <*digit-sequence*>
<digit-sequence>	::=	<*digit*>
	|	<*digit-sequence*> <*digit*>
<sign>	::=	+ | -
<integer-const>	::=	[<*sign*>] <*decimal-constant*> [<*suffix*>]
	|	[<*sign*>] <*hexadecimal-constant*> [<*suffix*>]
<decimal-const>	::=	<*nonzero-digit*>
	|	<*decimal-constant*> <*digit*>
<hexadecimal-const>	::=	0 X <*hex-digit*>
	|	<*hexadecimal-constant*> <*hex-digit*>
<nonzero-digit>	::=	1 | 2 | ... | 9

< hex-digit >	::=	any character from the set: 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
< suffix >	::=	l | L
< character-const >	::=	' < *c-char* > '
< c-char >	::=	any character except ' or \\
	|	< *escaped-character* >
< escaped-character >	::=	\' \" \n \ddd \xdd
< string-literal >	::=	" [< *s-char-sequence* >] "
< s-char-sequence >	::=	< *s-char* >
	|	< *s-char-sequence* > < *s-char* >
< s-char >	::=	any character except " \\ or \n
	|	< *escaped-character* >
< operator >	::=	One selected from the set: [ ] . = -
< punctuator >	::=	One selected from the set: [ ] : ;

APPENDIX E

Pseudocode

header_to_cdr()

```
Check if arguments passed in are legal.

If cdr version numbers are not compatible
   Error
Endif

Set up global variables with values passed into this routine--
   target_buffer pointer, octets available count, target language.

Initialize global vars for type attribute start and length, and
   total target buffer length used.

If a flag indicating whether the trailer_to_cdr() call hasn't been done
   prior to this call
   Error ("two header_to_cdr() calls were done without an intervening
          trailer_to_cdr() call")
       /* this is an "error" as opposed to a "warning" coupled with
          a call to trailer_to_cdr() since we can't tell if the current
          target_ptr buffer is a new one or the old one that didn't
          use a trailer_to_cdr() call.  So we'll just flag it as an
          error. */
Endif

If the target_buffer pointer is NULL
   Error
Endif

If target language is invalid
   Error
Endif

If the size of the target_buffer cannot contain the data buffer header
   Error
Endif

/* Assign values to the data buffer header in target_buffer--   */
/* "Hand-marshall" means you must do it directly without the    */
/* aid of an ASN.1 routine.                                     */

Hand-marshall version control id and target lang into the target_buffer.
```

```
If not okay
   Error
Endif

Hand-marshall reservation space for a 4-byte ASN.1 integer for the type
   attribute start.      /* NOTE: this is a non-standard ASN.1 integer */
If not okay  -   /* will fill in this value after marshalling is complete */
   Error
Endif

Hand-marshall reservation space for a 4-byte ASN.1 integer for the
   type attribute length.     /* NOTE: this is a non-standard ASN.1 integer */
If not okay      /* will fill in this value after marshalling is complete */
   Error
Endif

Hand-marshall the type/length octets of an octetstring into the temp_buffer
   to start the ASN.1 format for the type attributes.
Set last_item = not_string.

Allocate a default stack frame for array_stack.
Set top_of_stack fields to  rec_count = 0; init_already = no
```

trailer_to_cdr()

```
If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

If  (currently open octetstring in type attributes buffer)
    Put end-of-contents octets in type attribute buffer to close off currently
    open octetstring.
Endif

Concatenate temp_buffer onto the target_buffer by copying octets from
   the temp_buffer(s) to the target_buffer.

Update type_attr_start value in data buffer header with the final value.
Update type_attr_length value in data buffer header with the final value.

/* Optional Check:  was array_stack was emptied properly? */

Return to the caller the total number of octets used for marshalling
   for all routines.
```

array_to_cdr()

```
If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
```

```
      Allocate another temp_buffer and link it into the temp_buffer list.
   Endif

   Restart type attributes octetstring for IDs.
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

If top_of_stack init_already = no

   If compression = PACKED
      Assign type attribute value to temp_buffer.
   Else if compression = NONE
      Assign type attribute value to temp_buffer.
   Else
      Error
   Endif

Endif

Push new frame on array_stack.

Initialize new stack frame field rec_count to 0.
Initialize new stack frame field init_already to the init_already
   value of old top-of-stack.

Assign ASN.1 data element to target_buffer.
```

array_end_to_cdr()

```
Pop stack frame from array_stack.

If top_of_stack rec_count = 0   /* this was an array or array of array? */
   Set init_already = yes.
Endif    /* else the array that just ended was a part of a record and
            the record hasn't ended yet */

If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

Assign ASN.1 data element to target_buffer.
```

bit_field_to_cdr()

```
If source_ptr = NULL
   Error
Endif

If start_bit_position < 0 or > 31
   Error
Endif

If num_of_bits < 1 or > 32
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
      Restart type attributes octetstring for IDs.
   Endif
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

Calculate number of bytes needed to store bits.
If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

If top_of_stack init_already = no

   If signed = SIGNED
      Assign type attribute value to temp_buffer.
   Else if signed = UNSIGNED
      Assign type attribute value to temp_buffer.
   Else
      Error
   Endif

   If top_of_stack rec_count = 0   /* for the simple-type array type */
      Set init_already = yes.
   Endif   /* else this data element was a part of a record */
Endif

Assign ASN.1 data element to target_buffer.
```

bool_to_cdr()

```
If source_ptr = NULL
   Error
Endif

If bool_value <> CDR_TRUE and <> CDR_FALSE
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
      Restart type attributes octetstring for IDs.
   Endif
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

If top_of_stack init_already = no

   If DDL_type = SHORT_BOOLEAN
      Assign type attribute value to temp_buffer.
   Else if DDL_type = BOOLEAN
      Assign attribute value to temp_buffer.
   Else if DDL_type = LONG_BOOLEAN
      Assign type attribute value to temp_buffer.
   Else
      Error
   Endif

   If top_of_stack rec_count = 0   /* for the simple-type array type */
      Set init_already = yes.
   Endif   /* else this data element was a part of a record */
Endif

Assign ASN.1 data element to target_buffer.
```

char_to_cdr()

```
If source_ptr = NULL
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
      Restart type attributes octetstring for IDs.
   Endif
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

If top_of_stack init_already = no

   Assign type attribute ID to temp_buffer.

   If top_of_stack rec_count = 0   /* for the simple-type array type */
      Set init_already = yes.
   Endif   /* else this data element was a part of a record */
Endif

Assign ASN.1 data element to target_buffer.
```

int_to_cdr()

```
If source_ptr = NULL
   Error
Endif

If bit_length = 0
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
```

```
      Restart type attributes octetstring for IDs.
   Endif
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

Calculate number of bytes needed to store bits.
If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

If top_of_stack init_already = no

   If DDL_type = BYTE
      Assign type attribute value to temp_buffer.
   Else if DDL_type = SHORT_INT
      Assign attribute value to temp_buffer.
   Else if DDL_type = INTEGER
      Assign type attribute value to temp_buffer.
   Else if DDL_type = LONG_INT
      Assign type attribute value to temp_buffer.
   Else
      Error
   Endif

   If top_of_stack rec_count = 0   /* for the simple-type array type */
      Set init_already = yes.
   Endif   /* else this data element was a part of a record */
Endif

Assign ASN.1 data element to target_buffer.
```

opaque_to_cdr()

```
If source_ptr = NULL
   Error
Endif

If num_of_octets = 0
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
```

```
      Restart type attributes octetstring for IDs.
   Endif
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

If top_of_stack init_already = no

   Assign type attribute ID to temp_buffer.

   If top_of_stack rec_count = 0   /* for the simple-type array type */
      Set init_already = yes.
   Endif   /* else this data element was a part of a record */
Endif

Assign ASN.1 data element to target_buffer.

        ! real_to_cdr()
If source_ptr = NULL
   Error
Endif

If bit_length = 0
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
      Restart type attributes octetstring for IDs.
   Endif
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

Calculate number of bytes needed to store bits.
If there is not enough space in the target_buffer for this ASN.1 data element
```

```
   Error
Endif

If top_of_stack init_already = no

   If DDL_type = REAL
      Assign type attribute value to temp_buffer.
   Else if DDL_type = LONG_REAL
      Assign type attribute value to temp_buffer.
   Else
      Error
   Endif

   If top_of_stack rec_count = 0   /* for the simple-type array type */
      Set init_already = yes.
   Endif   /* else this data element was a part of a record */
Endif

Assign ASN.1 data element to target_buffer.
```

record_to_cdr()

```
If restrictve_type does not have a legal value
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
      Restart type attributes octetstring for IDs.
   Endif
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   and the restrictive_alignment type attribute
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

If init_already = no

   If compression = PACKED
      Assign type attribute value to temp_buffer.
   Else if compression = NONE
```

```
      Assign type attribute value to temp_buffer.
   Else
      Error
   Endif

   Assign type attribute ID to temp_buffer.

   Increment top_of_stack rec_count.
Endif

Assign restrictive_alignment type attribute to temp_buffer.

Assign ASN.1 data element to target_buffer.
```

record_end_to_cdr()

```
If init_already = no
   Decrement top_of_stack rec_count.
   If top_of_stack rec_count = 0    /* must do this check here for   */
      Set init_already = yes.       /* array of records to work      */
   Endif
Endif

If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

Assign ASN.1 data element to target_buffer.
```

\13 string_to_cdr()

```
If source_ptr = NULL
   Error
Endif

If max_num_chars = 0   OR
   curr_length > max_num_chars
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
      Restart type attributes octetstring for IDs.
   Endif
Endif

If there is not enough space in the temp_buffer for this type attribute ID
```

```
   and max_num_chars
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

If there is not enough space in the target_buffer for this ASN.1 data element
   Error
Endif

If top_of_stack init_already = no

   Assign type attribute ID to temp_buffer.

   End temp_buffer type attributes octetstring.
   Assign max_num_chars type attribute to temp_buffer.

   If top_of_stack rec_count = 0   /* for the simple-type array type */
      Set init_already = yes.
   Endif   /* else this data element was a part of a record */
Endif

Assign ASN.1 data element to target_buffer.

Set last_item = string.
```

u_int_to_cdr()

```
If source_ptr = NULL
   Error
Endif

If bit_length = 0
   Error
Endif

If last_item was string type
   If there is not enough space in the temp_buffer to restart octetstring
      in type attributes buffer
      Allocate another temp_buffer and link it into the temp_buffer list.
   Else
      Restart type attributes octetstring for IDs.
   Endif
   Set last_item = not_string.
Endif

If there is not enough space in the temp_buffer for this type attribute ID
   Allocate another temp_buffer and link it into the temp_buffer list
Endif

Calculate number of bytes needed to store bits.
If there is not enough space in the target_buffer for this ASN.1 data element
```

```
   Error
Endif

If top_of_stack init_already = no

   If DDL_type = U_BYTE
      Assign type attribute value to temp_buffer.
   Else if DDL_type = U_SHORT_INT
      Assign attribute value to temp_buffer.
   Else if DDL_type = U_INTEGER
      Assign type attribute value to temp_buffer.
   Else if DDL_type = U_LONG_INT
      Assign type attribute value to temp_buffer.
   Else
      Error
   Endif

   If top_of_stack rec_count = 0   /* for the simple-type array type */
      Set init_already = yes.
   Endif   /* else this data element was a part of a record */
Endif

Assign ASN.1 data element to target_buffer.
```

Test Cases for Type Attribute Compaction

These test cases show the type attribute compaction that would be performed for the data structures specified. The asterisked ('*') items indicate the type attribute IDs that would actually go into the type attributes buffer:

1. **Testing:** simple data type

   Marshalling Calls Used:

```
int *
```

2. **Testing:** array of simple type

   Marshalling Calls Used:

```
array *
   int *
   int
   int
array_end
```

3. **Testing:** record containing a simple type

   Marshalling Calls Used:

```
record *
   int *
record_end
```

4. **Testing:** record containing two simple types

   Marshalling Calls Used:

```
record *
   int *
   char *
record_end
```

5. **Testing:** record containing an array of simple type

   Marshalling Calls Used:

```
record *
   array *
      int *
      int
      int
   array_end
record_end
```

6. **Testing:** record containing an array of simple type followed by a different data element

Marshalling Calls Used:

```
record *
   array *
      int *
      int
      int
   array_end
   int *
record_end
```

7. **Testing:** record containing arrays of simple type

Marshalling Calls Used:

```
record *
   array *
      int *
      int
      int
   array_end
   array *
      int *
      int
      int
   array_end
record_end
```

8. **Testing:** array of records

Marshalling Calls Used:

```
array *
   record *
      int *
   record_end
   record
      int
   record_end
   record
      int
   record_end
array_end
```

9. **Testing:** nested records

Marshalling Calls Used:

```
record *
   int *
   record *
      int *
   record_end
   int *
record_end
```

10. **Testing:** array of array of simple type, version 1

Marshalling Calls Used:

```
array *
   array *
      int *
      int
      int
   array_end
array_end
```

11. **Testing:** array of array of simple type, version 2

Marshalling Calls Used:

```
array *
   array *
      int *
      int
      int
   array_end
   array
      int
      int
      int
   array_end
   array
      int
      int
      int
   array_end
array_end
```

12. **Testing:** array of array of array of simple type

Marshalling Calls Used:

```
array *
   array *
      array *
         int *
         int
      array_end
   array_end
   array
      array
         int
         int
      array_end
   array_end
array_end
```

APPENDIX F

Pseudocode

Library Initialization

```
/* Done at compile/link time:      */

Set pointers in language tables to point to low-level unmarshalling routines.

Set pointers in language table pointer (ltp) table (ltp_table)
   to point to language tables.
```

Main Unmarshalling Routine

```
/*
   Ready to start processing next data message that comes in.  Call
   the sequence of code below for each new message that comes in.

   target_ptr is passed in from the caller and should already be
   pointing to the start of the target_buffer in which unmarshalled
   data elements will be deposited.

   The same will be true of octets_avail.  This will be passed in by
   the caller and will be initialized to the number of bytes in the
   target_buffer.
*/

bits_avail = 0.  /* used in case unmarshalled elements take up only parts
                    of bytes */

Set data_buf_ptr to the start of the data buffer (set equal to source_ptr).

Unmarshall the data buffer header in the source_buffer using a method that
   doesn't use type attributes.  Main concern here is maintaining
   consistency in type and size of header data between machines.
   Return pointer to remainder of source_buffer (the start of the user data).

Set source_ptr to the start of the user data (use the pointer
   returned from above).

Use language ID from the source_buffer data buffer header as an offset
   into ltp_table.  Set ltp_table_ptr to point to this location in
   ltp_table.

Add type_attr_start offset to data_buf_ptr to obtain address of start
   of type attributes area.  Set type_attr_ptr to this address.

Add type_attr_length to type_attr_start and assign to end_of_type_attr_ptr.
   end_of_type_attr_ptr offers another way to check for end-of-processing
   (= whole message has been unmarshalled).  end_of_type_attr_ptr should
```

```
   be pointing to the byte address AFTER the last type attribute byte.

Assign end_of_user_data_ptr to type_attr_ptr.  end_of_user_data_ptr
   offers still another way to check for end-of-processing (= whole
   message has been unmarshalled).  end_of_user_data_ptr should
   be pointing to the byte address AFTER the last user data byte (which
   is the same as the start of the type attributes).

top_of_stack = NULL              /* start with a fresh stack */

Push a "default" stack frame onto the stack.  type_attribute field
   of the stack frame is "default".  next_item_incr field is set to 1.
   This default stack frame is used for processing data messages
   containing simple data types.

open_octetstring = false.   /* flag used in checking type attr completion */

Repeat      /* do this loop once for each user data item */

   If open_octetstring = false   /* start new octetstring */
      If type_attr_ptr points to octetstring type/length
         Move type_attr_ptr past octetstring type/length so that it
            points to the next type attribute to process.
      Else
         Error
      Endif
   Else   /* still in the middle of an open octetstring */
      If type_attr_ptr points to octetstring type/length
         Error
      Endif
   Endif

   /* at this point, type_attr_ptr should be pointing to the next
      type attribute to process */

   open_octetstring = true.  /* the header for the type attributes octetstring
                                has been read; flag means octetstring closing
                                trailer will eventually be expected to show
                                up */

   Verify that the type attribute ID is correctly paired with the user data
      ASN.1 tag type by looking up the type attribute in the ddl_asn1_table
      and retrieving the associated ASN.1 tag.

   If the user data tag = ddl_asn1_table tag  /* match between user data
                                                 and type attributes?   */

      If the type attribute ID is an array/packed-array/record/packed-record
         If bits_avail <> 0   /* align target_ptr to next byte boundary */
            target_ptr = target_ptr + 1.
```

```
      bits_avail = 0
   Endif

   If type attribute ID is an array or packed array
      Allocate a stack frame.
      If (Pascal/300)
         /* Set top_of_stack values: */
         2-byte_align_ptr = Calculate 2-byte alignment.
         4-byte_align_ptr = Calculate 4-byte alignment.
         target_ptr       = 4-byte_align_ptr. /* most possible case */
      Endif
      /* else let element type set the alignment for the array */

      Set next_item_incr in the stack frame to 0.
      Increment type_attr_ptr by adding hardcoded 1 to it.
   Else
      If (Pascal/300)
         Allocate a stack frame.
         /* Set top_of_stack values: */
         2-byte_align_ptr = Calculate 2-byte alignment.
         4-byte_align_ptr = Calculate 4-byte alignment.
         target_ptr       = 4-byte_align_ptr. /* most possible case */
      Else If (Pascal/800)
         Allocate a stack frame.
         Align to largest field or byte, whichever is larger.
      Else If (C/300)
         If top_of_stack = default  /* stack empty? */
            Alignment is 4 bytes.
         Else
            Alignment is 2 bytes.
         Endif
         Allocate a stack frame.
      Else If (C/800)
         Allocate a stack frame.
         Align the target_ptr to a suitable starting location according
            to most_restrictive field in stack frame.
      Endif
      /* else for Fortran/300, Fortran/800 let low-level routine
         do the aligning */

      Set next_item_incr field in the stack frame to 1.
      Increment type_attr_ptr by adding hardcoded 2 to it.
   Endif

   Increment source_ptr to next data element.

Else  /* not a structured type, just a simple one...  */

   /* need to feed size and alignment to the particular unmarshall
      routine; these are passed as globals to the routine:   */
```

```
                size = (*(ltp_table[ltp_table_ptr][top_of_stack->packed_or_not]))
                          [*type_attr_ptr][size];
                alignment = (*(ltp_table[ltp_table_ptr][top_of_stack->packed_or_not]))
                          [*type_attr_ptr][alignment];

                If size or alignment are <= 0   /* which they shouldn't be; */
                   Warning                      /* means lang tables are corrupt */
                   Use appropo defaults.        /* or incorrect. */
                Endif

                /* Call an ASN.1 low-level unmarshall routine specific to the
                   DDL type; this routine is pointed to by the DDL entry in the
                   language table.

                   Unlike the other data types, the cdr_to_string() routine will
                   read the type attributes end-octetstring.  It will then read the
                   max_num_chars type attribute.  After reading the end-octetstring,
                   the routine sets open_octetstring to false.

                   All routines increment source_ptr to the next data element
                   only if successful; do not increment if not successful.
                */

                If ((result = ltp_table_ptr->[stack->packed]lt_ptr->
                      lang_table[*type_attr_ptr]->unmarsh_rtn()) != SpSuccess)
                   Error
                Endif

                If top_of_stack next_item_incr is 1
                   Set type_attr_ptr to type_attr_last_pos.
                Endif  /* else current structured type is an array so don't move
                          type_attr_ptr */

                /* type_attr_last_pos is the furthest position in the type
                   attribute buffer that type_attr_ptr has travelled. */

             Endif

          Endif

          If source_ptr < end_of_user_data_ptr  /* if any user data remains */

             /* eoc processing : attempt to advance source_ptr before advancing
                type_attr_ptr */

             While (source_ptr octet = eoc) DO  /* may be many eoc's in a row */

                Increment source_ptr to next data item.
```

```
            /* need to adjust alignment of 300 structured type? */
            If (Pascal/300)

               /* looking for the one-/two-byte structures */
               If ((size_of_struct = target_ptr - 4-byte_align_ptr) <= 2)

                  /* one-byte case? */
                  If (top_of_stack = record) AND (size_of_struct = 1) AND
                     (1-byte_align_ptr <> 4-byte_align_ptr) /* same location? */
                     Byte-copy size_of_struct number of bytes from 4-byte_align_ptr
                        to 1-byte_align_ptr location.
                     target_ptr = target_ptr - 1

                  /* two-byte case? */
                  Else If (2-byte_align_ptr <> 4-byte_align_ptr)
                     Byte-copy size_of_struct number of bytes from 4-byte_align_ptr
                        to 2-byte_align_ptr location.
                     target_ptr = target_ptr - 2
                  Endif
               Endif
            Endif

            If (C/800) AND (top_of_stack is a record)
               Align target_ptr so that next unmarshalled data element
                  starts on a multiple of most-restrictive type in record
                  (need to check this multipleness alignment
                  business with Pascal).
            Endif
            If top_of_stack = default   /* stack empty too soon? */
               Error
            Endif
            If top_of_stack = array .
               Set type_attr_ptr to type_attr_last_pos.
            Endif
            Pop top_of_stack
            If new top_of_stack = default    /* stack empty? */
               Break loop.
            Endif

            If new top_of_stack is array
               Set type_attr_ptr to array_start_pos.
                  /* if this top_of_stack array frame was for an array of arrays
                     or an array of records, type_attr_ptr might have
                     incremented far from it.  Need to reset back to handle
                     next array/record within the array */
            Endif

         Endwhile;

      Endif
```

```
Until (top_of_stack = default) OR
      (source_ptr = end_of_user_data_ptr) OR
      (type_attr_ptr = end_of_type_attr_ptr);

/* check type-attribute completion depending on final state */

If open_octetstring = true  /* final type attribute eoc was not read? */
   If type_attr_ptr + 2 <> end_of_type_attr_ptr
      Warning
   Else if type_attr_ptr is not an eoc
      Warning
   Endif
Else
   If type_attr_ptr <> end_of_type_attr_ptr
      Warning
   Endif
Endif

/* Do some more checks to see if processing was totally complete */

If source_ptr <> end_of_user_data_ptr  /* some user data left   */
   Error                               /* unprocessed?         */
Else If top_of_stack <> default    /* did not close some records/arrays? */
   Error
Endif

/* reclaim stack space */

Free (top_of_stack).
```

Primitive Unmarshalling Routines

There are pointers in the Language Tables to these routines. During the unmarshalling process, a Language Table is accessed and the pointer to the specific primitive unmarshalling routine is used to invoke that routine. At that point, the actual process of unmarshalling takes place. The routine uses parameters from the Language Table itself to do the unmarshalling. These parameters include target data element size and alignment. After the conversion is performed, the primitive routine will deposit the converted data element in the target_buffer.

These routines are as follows:

cdr_to_header()

unmarshalling shell

cdr_to_trailer()

unmarshalling shell

cdr_to_bit_field()

```
/* Temporaries for user_data_ptr, target_ptr */
sp_temp = user_data_ptr + 1; (ASN type field checked in shell)
tp_temp = target_ptr;

/* Calculate alignment */
Update tp_temp according to alignment.

/* Check for available space */
If  (size > space left after appropriate alignment)
    Error( ERR_OUTOFBUFFER )
Endif

length <- ASN.1 length field (must be >=1, <=9), updating sp_temp
unused <- ASN.1 contents octet one (must be >=0, <=7), updating sp_temp
If (size <  ((length-1)*8 - unused))
    Error( ERR_TOOBIG )
Endif

Loop on ASN.1 contents octets two to length
    If  ((last (length) octet) and (unused <> 0))
                                Transfer bits, updating sp_temp, tp_temp
    Else
                                Transfer byte, updating sp_temp, tp_temp
    Endif
Endloop

Update octets_avail, bits_avail
```

```
Update user_data_ptr from sp_temp.
Update target_ptr from tp_temp.
Return( success ).
```

cdr_to_bool()

```
/* Temporaries for user_data_ptr, target_ptr */
sp_temp = user_data_ptr + 1; (ASN type field checked in shell)
tp_temp = target_ptr;

/* Calculate alignment */
Update tp_temp according to alignment.

/* Check for available space */
If  (size > space left after appropriate alignment)
    Error( ERR_OUTOFBUFFER )
Endif

Check ASN.1 length field (should be 1), updating sp_temp.

If  (ASN contents octet == 0)
    assign FALSE to target (bit or byte), updating sp_temp, tp_temp
Else
    assign TRUE to target (bit or byte), updating sp_temp, tp_temp
Endif

Update octets_avail, bits_avail
Update user_data_ptr from sp_temp.
Update target_ptr from tp_temp.
Return( success ).
```

cdr_to_char()

```
/* Temporaries for user_data_ptr, target_ptr */
sp_temp = user_data_ptr + 1; (ASN type field checked in shell)
tp_temp = target_ptr;

/* Calculate alignment */
Update tp_temp according to alignment.

/* Check for available space */
If  (size > space left after appropriate alignment)
    Error( ERR_OUTOFBUFFER )
Endif

Check ASN.1 length field (should be 1), updating sp_temp.

If (size < 8)
```

```
    Error( ERR_TOOBIG )
Endif

Transfer the byte (assume ASCII), updating sp_temp, tp_temp.

Update octets_avail, bits_avail
Update user_data_ptr from sp_temp.
Update target_ptr from tp_temp.
Return( success ).
```

cdr_to_int()

```
/* Temporaries for user_data_ptr, target_ptr */
sp_temp = user_data_ptr + 1; (ASN type field checked in shell)
tp_temp = target_ptr;

/* Calculate alignment */
Update tp_temp according to alignment.

/* Check for available space */
If  (size > space left after appropriate alignment)
    Error( ERR_OUTOFBUFFER )
Endif

length <- ASN.1 length field, updating sp_temp.

If (size < 8*length)
    Error( ERR_TOOBIG )
Endif

Transfer the bytes (sign extend if necessary), updating sp_temp, tp_temp.

Update octets_avail, bits_avail
Update user_data_ptr from sp_temp.
Update target_ptr from tp_temp.
Return( success ).
```

cdr_to_opaque()

```
/* Temporaries for user_data_ptr, target_ptr */
sp_temp = user_data_ptr + 1; (ASN type field checked in shell)
tp_temp = target_ptr;

/* Calculate alignment */
Update tp_temp according to alignment.

/* Check for available space */
If  (size > space left after appropriate alignment)
```

```
    Error( ERR_OUTOFBUFFER )
Endif

length <- ASN.1 length field, updating sp_temp.

If (size < 8*length)
    Error( ERR_TOOBIG )
Endif

Transfer the bytes (octets), updating sp_temp, tp_temp.

Update octets_avail, bits_avail
Update user_data_ptr from sp_temp.
Update target_ptr from tp_temp.
Return( success ).
```

cdr_to_real()

```
/* Temporaries for user_data_ptr, target_ptr */
sp_temp = user_data_ptr + 1; (ASN type field checked in shell)
tp_temp = target_ptr;

/* Calculate alignment */
Update tp_temp according to alignment.

/* Check for available space */
If  (size > space left after appropriate alignment)
    Error( ERR_OUTOFBUFFER )
Endif

length <- ASN.1 length field, updating sp_temp.

If (size < format_dependent_value)
    Error( ERR_TOOBIG )
Endif

Transfer the real (involved translation), updating sp_temp, tp_temp.

Update octets_avail, bits_avail
Update user_data_ptr from sp_temp.
Update target_ptr from tp_temp.
Return( success ).
```

cdr_to_string()

```
/* Temporaries for user_data_ptr, target_ptr, type_attr_ptr */
sp_temp = user_data_ptr + 1; (ASN type field checked in shell)
```

```
tp_temp = target_ptr;
ap_temp = type_attr_ptr + 1; (shell saw type attribute: string)

/* process type attribute information: eoc, ASN.1 integer (maxlen) */
If  ((ap_temp->)++ <> eoc)
    Warning(7)
Endif
If  ((ap_temp->)++ <> type int)
    Warning(7)
Endif
maxlensize = (ap_temp->)++
Transfer maxlen <- (ap_temp->)[maxlensize bytes], updating ap_temp


/* Calculate alignment */
Update tp_temp according to alignment.

/* Check for available space */
If  (size > space left after appropriate alignment)
    Error( ERR_OUTOFBUFFER )
Endif

length <- ASN.1 length field, updating sp_temp.

If  (length > maxlen)
    Warning( truncation )
Endif

If (size < maxlen)
    Error( ERR_TOOBIG )
Endif

Transfer the bytes (assume ASCII), updating sp_temp, tp_temp (by maxlen ?)
    Null-terminate (C) or blank pad, if necessary.

Update octets_avail, bits_avail
Update user_data_ptr from sp_temp.
Update target_ptr from tp_temp.
Update type_attr_ptr from ap_temp.
Set open_octetstring to FALSE.
Return( success ).
```

cdr_to_u_int()

```
/* Temporaries for user_data_ptr, target_ptr */
sp_temp = user_data_ptr + 1; (ASN type field checked in shell)
tp_temp = target_ptr;

/* Calculate alignment */
```

```
Update tp_temp according to alignment.

/* Check for available space */
If  (size > space left after appropriate alignment)
    Error( ERR_OUTOFBUFFER )
Endif

length <- ASN.1 length field, updating sp_temp.

If (size < 8*length)
    Error( ERR_TOOBIG )
Endif

Transfer the bytes (zero extend if necessary), updating sp_temp, tp_temp.

Update octets_avail, bits_avail
Update user_data_ptr from sp_temp.
Update target_ptr from tp_temp.
Return( success ).
```

Test Cases for Main Unmarshalling Routine

The following is a listing of test cases used to desk check the data buffer parsing logic of the main routine pseudocode.

For the user data specifications below, each item between commas represents one ASN.1 data element.

For the type attributes specifications below, each item between commas represents one physical octet.

1. Testing: simple data type

   Logical Format:

   ```
   char
   ```

   user data: char

   type attributes: octetstring type, length, char ID, eoc type, length

2. Testing: slight more internally complex simple type

   Logical Format:

   ```
   string
   ```

   user data: string

   type attributes: octetstring type, length, string ID, eoc type, length, int type, length, contents (max string size)

3. Testing: array with simple type

   Logical Format:

   ```
   array [3] of char
   ```

   user data: seq-of, char, char, char, eoc

type attributes: octetstring type, length, array ID, char ID, eoc type, length

4. Testing: array with slightly more internally complex simple type

Logical Format:

```
array [3] of string
```

user data: seq-of, string, string, string, eoc

type attributes: octetstring type, length, array ID, string ID, eoc type, length, int type, length, contents (max string size)

5. Testing: record containing a simple type

Logical Format:

```
record
   char
end
```

user data: seq, char, eoc

type attributes: octetstring type, length, record ID, restrictive type, char ID, eoc type, length

6. Testing: record containing two simple types

Logical Format:

```
record
   bool
   char
end
```

user data: seq, boolean, char, eoc

type attributes: octetstring type, length, record ID, restrictive type, boolean ID, char ID, eoc type, length

7. Testing: record containing string type

Logical Format:

```
record
   string
end
```

user data: seq, string, eoc

type attributes: octetstring type, length, record ID, restrictive type, string ID, eoc type, length, int type, length, contents (max string size)

8. Testing: record containing string type and simple type

Logical Format:

```
record
   string
   char
end
```

user data: seq, string, char, eoc

type attributes: octetstring type, length, record ID, restrictive type, string ID, eoc type, length, int type, length, contents (max string size), octetstring type, length, char ID, eoc type, length

9. Testing: record containing an array of simple type

Logical Format:

```
record
   array [2] of char
end
```

user data: seq, seq-of, char, char, eoc, eoc

type attributes: octetstring type, length, record ID, restrictive type, array ID, char ID, eoc type, length

10. Testing: record containing an array of simple type followed by a different data element

Logical Format:

```
record
   array [2] of boolean
   char
end
```

user data: seq, seq-of, boolean, boolean, eoc, char, eoc

type attributes: octetstring type, length, record ID, restrictive type, array ID, boolean ID, char ID, eoc type, length

11. Testing: record containing an array of string type

Logical Format:

```
record
   array [2] of string
end
```

user data: seq, seq-of, string, string, eoc, eoc

type attributes: octetstring type, length, record ID, restrictive type, array ID, string ID, eoc type, length, int type, length, contents

12. Testing: record containing an array of string type followed by a different data element

Logical Format:

```
record
   array [2] of string
   char
end
```

user data: seq, seq-of, string, string, eoc, char, eoc

type attributes: octetstring type, length, record ID, restrictive type, array ID, string ID, eoc type, length, int type, length, contents, octetstring type, length, char ID, eoc type, length

13. Testing: array of records

Logical Format:

```
array [2] of record
                char
                boolean
             end
```

user data: seq-of, seq, char, boolean, eoc, seq, char, boolean, eoc, eoc

type attributes: octetstring type, length, array ID, record ID, restrictive type, char ID, boolean ID, eoc type, length

14. Testing: ne d records

Logical Format:

```
record
   char
   record
      char
   end
   char
end
```

user data: seq, char, seq, char, eoc, char, eoc

type attributes: octetstring type, length, record ID, restrictive type, char ID, record ID, restrictive type, char ID, char ID, eoc type, length

15. Testing: array of packed array of simple type

Logical Format:

```
array [3] of packed array [2] of char
```

user data: seq-of, seq-of, char, char, eoc, seq-of, char, char, eoc, seq-of, char, char, eoc, eoc

type attributes: octetstring type, length, array ID, packed array ID, char ID, eoc type, length

16. Testing: packed array of array of simple type

Logical Format:

```
packed array [3] of array [2] of char
```

user data: seq-of, seq-of, char, char, eoc, seq-of, char, char, eoc, seq-of, char, char, eoc, eoc

type attributes: octetstring type, length, packed array ID, array ID, char ID, eoc type, length

17. Testing: array of array of string type

Logical Format:

```
array [3] of array [2] of string
```

user data: seq-of, seq-of, string, string, eoc, seq-of, string, string, eoc, seq-of, string, string, eoc,

eoc

type attributes: octetstring type, length, array ID, array ID, string ID, eoc type, length, int type, length, contents (max string size)

18. Testing: array of array of record type

Logical Format:

```
array [3] of array [2] of record
                              char
                            end
```

user data: seq-of, seq-of, seq, char, eoc, seq, char, eoc, eoc, seq-of, seq, char, eoc, seq, char, eoc, eoc, seq-of, seq, int, eoc, seq, int, eoc, eoc, eoc

type attributes: octetstring type, length, array ID, array ID, record ID, restrictive type, char ID, eoc type, length

19. Testing: record containing an array of records of simple type

Logical Format:

```
record
   array [2] of record
                  char
                end
end
```

user data: seq, seq-of, seq, char, eoc, seq, char, eoc, eoc, eoc

type attributes: octetstring type, length, record ID, restrictive type, array ID, record ID, restrictive type, char ID, eoc type, length

Claims

1. A system for integrating user software application programs (402) operating on a network (LAN), comprising:

application adapting means (404) operating on at least one node (400) of said network (LAN) for providing said user software application programs (402) with communicative access to said network (LAN);

message management means (406) at at least one node (400) of said network (LAN) for managing data transfer requests between said user software application programs (402) operating on nodes (400) and other software application programs (402) connected to said network (LAN);

means (416, 418) for establishing node to node communications between a source node (400) having a source user software application program (402) operating thereon from which a data transfer request has originated and a destination node (400) having a destination user software application program (402) operating thereon to which data is to be transferred; and

data manipulating means (412) at said nodes (400) for manipulating message data from said source user software application program (402) into a common data representation for transmission to said destination user software application program (402), said message data from said source user application program (402) being manipulated when at least one of the data types, data formats and data representations of said source and destination user software application programs (402) do not correspond to each other.

2. A system as in claim 1, further comprising means (418) at said nodes (400) for manipulating message data in said common data representation received from said network (LAN) into the data types, data formats and data representations of said nodes (400) when said destination user software application program (402) operates on said nodes (400).

3. A system as in claim 2, wherein said data manipulation means (412) manipulates message data from said source user software application program (402) into said common data representation when said source and destination user software application programs (402) are written in different computer programming languages.

4. A system as in claim 3, wherein said common data representation is independent of the architecture of the nodes (400) and computer programming languages used on said network (LAN).

5. A system as in claim 1, further comprising means (FIG. 5) for forming node-specific data manipulation means (528) at each node (400) for manipulating data from said source user software application program (402) into a common data representation for transmission, and for manipulating data received in said common data representation into data compatible with said destination user software application (402), said data manipulation forming means comprising:

file means (502) for storing a high level description of user software application programs (402) and nodes (400) operating on said network (LAN), the characteristics of data at each source and destination user software application program (402), and the manipulations necessary to convert data from source to destination characteristics;

validation module means (504) for generating, as source code on a node (400) designated as an administration node (NODE 1), configuration files (510), based on said high level description;

configuration table compiling means (506) and data manipulation compiling means (508) for generating, as source code on said administration node (NODE 1), manipulation files (512), based on said high level description;

data manipulation module builder means (520) for copying said manipulation files (512) and compiling, on each node (400) designated as a compilation node (NODE 2), said manipulation files (512) to form node-specific data manipulation modules (528); and

start up module means (516) for copying said configuration files (510), for loading said configuration files (510) in memory and for starting up said data manipulation files (512).

6. A method of forming a data manipulation module (528) for integrating user software application programs (402) operating on a network (LAN) so as to allow messages to be transmitted from a source user software application program (402) of a first data type to a destination user application program (402) of a second data type, comprising the steps of:

(a) storing (502) as configuration source code (510) a high-level description of user software application programs (402) and nodes (400) operating on said network (LAN);

(b) storing (508) as manipulation source code (512) the characteristics of data for source and destination user software application program (402) and the manipulations necessary to convert from data having the characteristics of data for a source user software application program (402) to data having the characteristics of data for a destination user software application program (402);

(c) compiling, at a compilation node (NODE 2), for each node type in said network (LAN), said configuration source code (510) and said manipulation source code (512) so as to form node-specific data manipulation modules (528) at each compilation node (NODE 2); and

(d) distributing said node-specific data manipulation modules (528) to each corresponding node (NODES 3-6) in distributed processing network.

7. A method as in Claim 6, wherein said compilation site is a node on said network and wherein said steps (a)-(d) are conducted during startup of said nodes (400) on said network (LAN), whereby said node-specific data manipulation modules (528) are used during run time of said network (LAN) to convert messages from said first data type to said second data type to complete transmission of said message from said source user software application program (402) to said destination user software application program (402).

8. A method as in claim 7, comprising the further step of validating (504) said configuration source code (510) prior to said startup step.

9. A method of transmitting message data from a first user application program (402) in a first programming language operating on a first node (CPU1) supporting a first data format to a second user application program (402) in a second programming language operating on a second node (CPU2) supporting a second data format, comprising the steps of:

generating a request from said first user application program (402) that said message data be sent to said second user application program (402);

determining whether a manipulation needs to be performed to send said message data from said first user application program (402) to said second user application program (402) so that said second user application program (402) can understand said message data;

if a manipulation needs to be performed, converting said message data to a common data representation independent of said first and second programming languages and said first and second data formats;

transmitting said message data from said first node (CPU1) to said second node (CPU2);

receiving said message data from said first node (CPU1) at said second node (CPU2);

determining whether a manipulation has been performed on said message data; and

if a manipulation has been performed on said message data, converting said message data from said common data representation thereof to said second programming language with said second data format.

10. A method as in claim 9, wherein said first and second user application programs (402) operate on the same node and hence only require programming language manipulations.

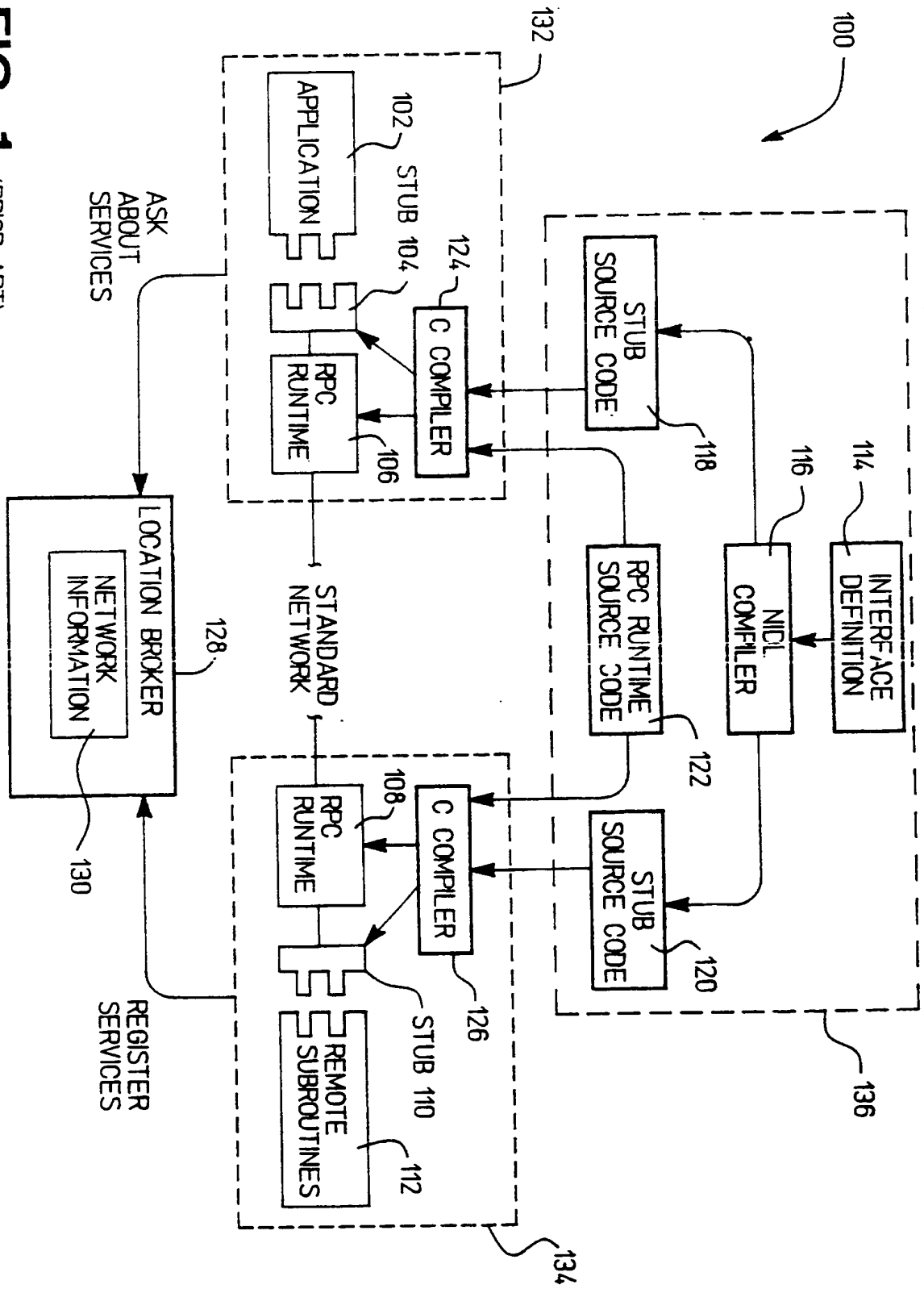


FIG 1 (PRIOR ART)

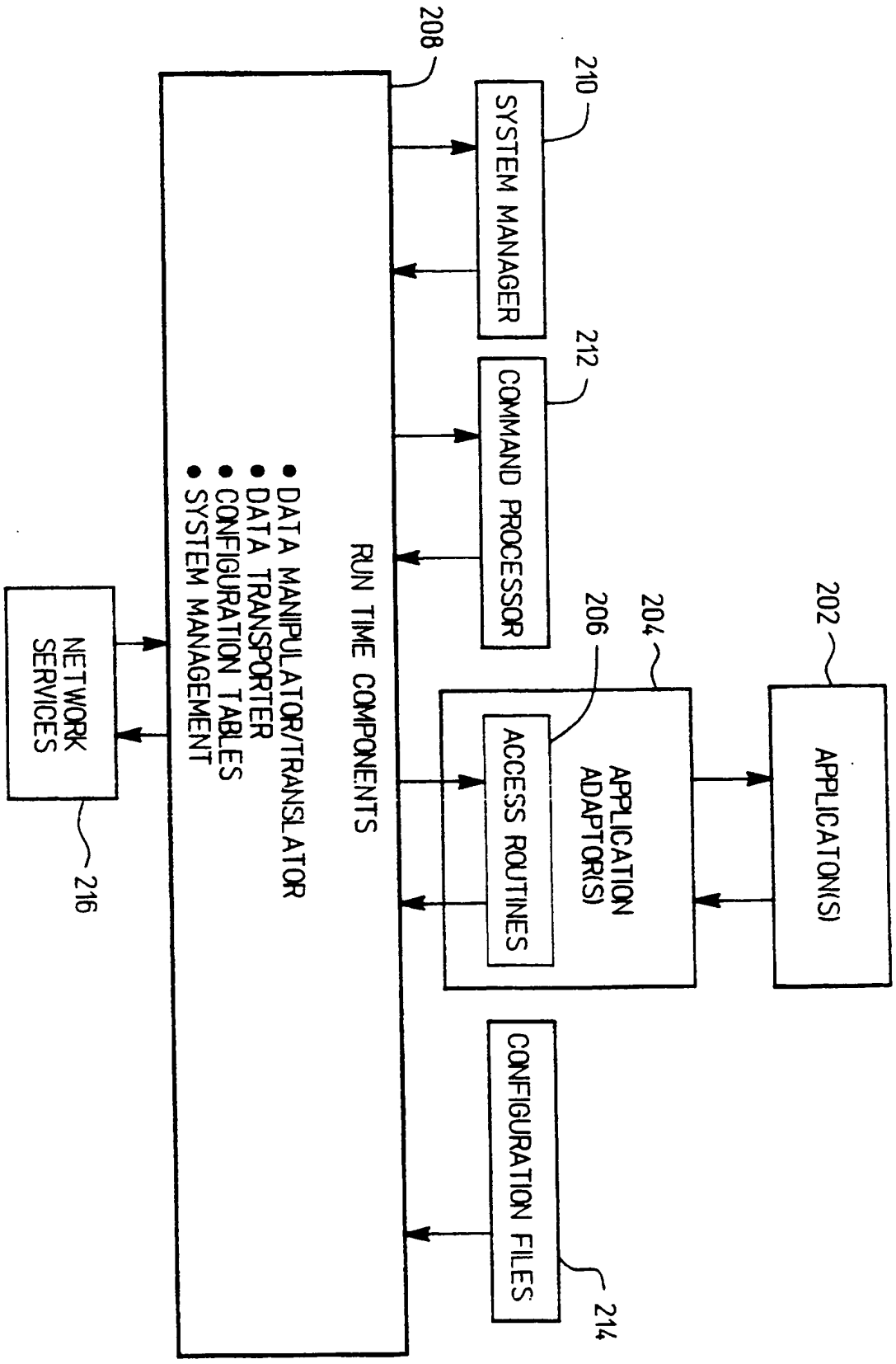


FIG 2

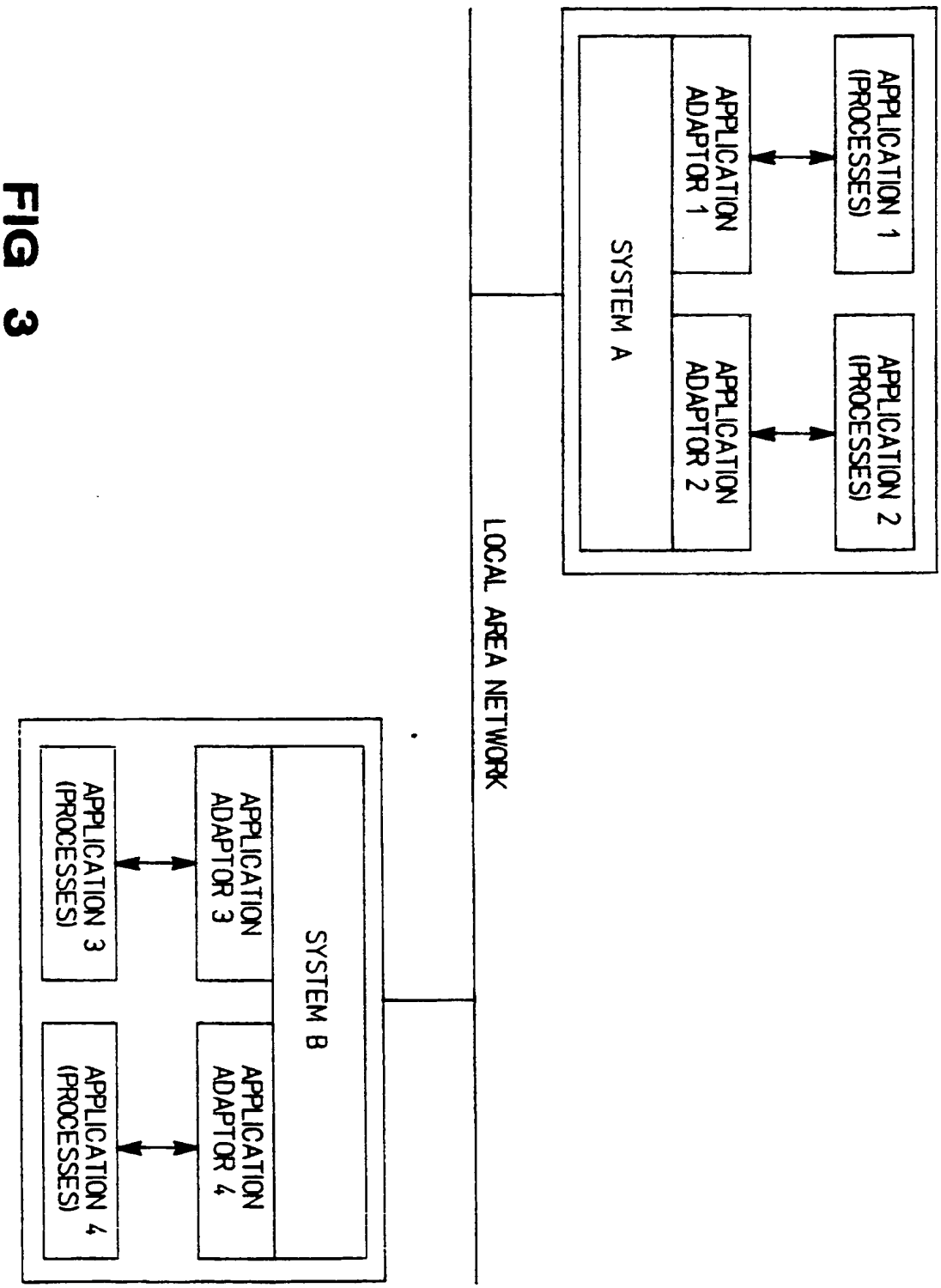


FIG 3

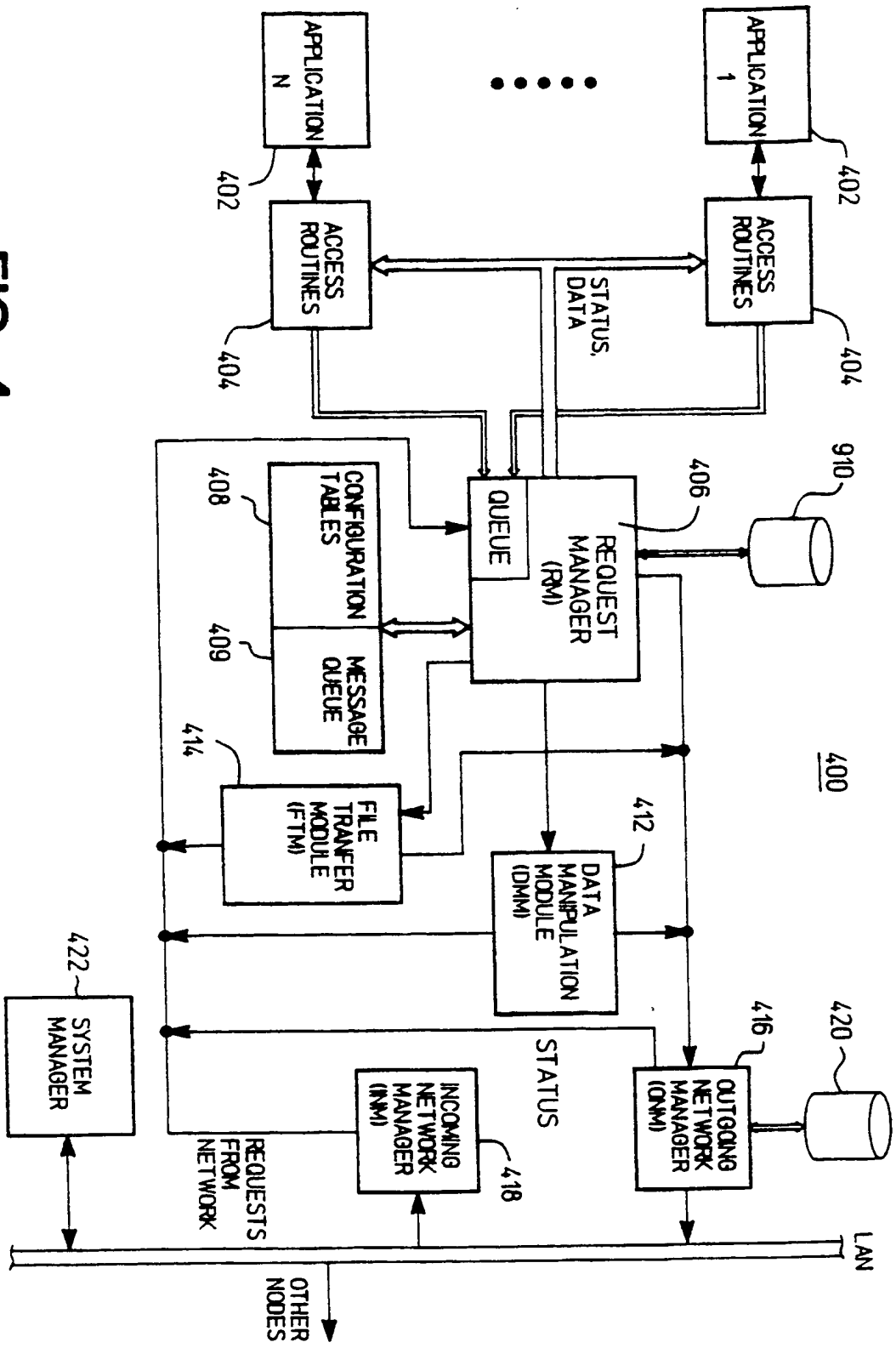

APPLICATION 1
APPLICATION N
402
ACCESS ROUTINES
404
STATUS, DATA
910
406
QUEUE
REQUEST MANAGER (RM)
408
CONFIGURATION TABLES
409
MESSAGE QUEUE
414
FILE TRANFER MODULE (FTM)
412
DATA MANIPULATION MODULE (DMM)
400
422
SYSTEM MANAGER
416
420
OUTGOING NETWORK MANAGER (ONM)
STATUS
INCOMING NETWORK MANAGER (INM)
418
REQUESTS FROM NETWORK
LAN
OTHER NODES

FIG 4

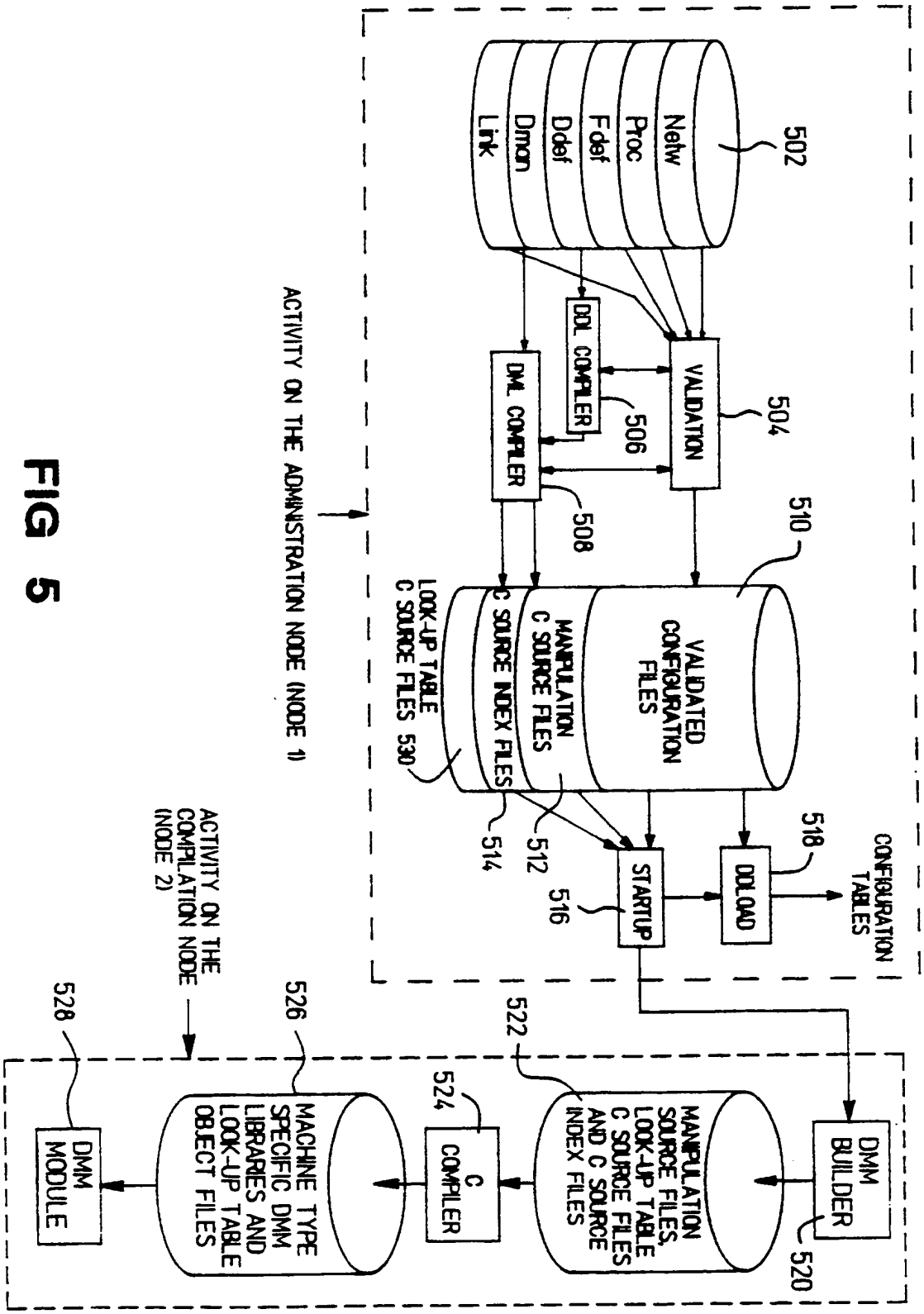


FIG 5

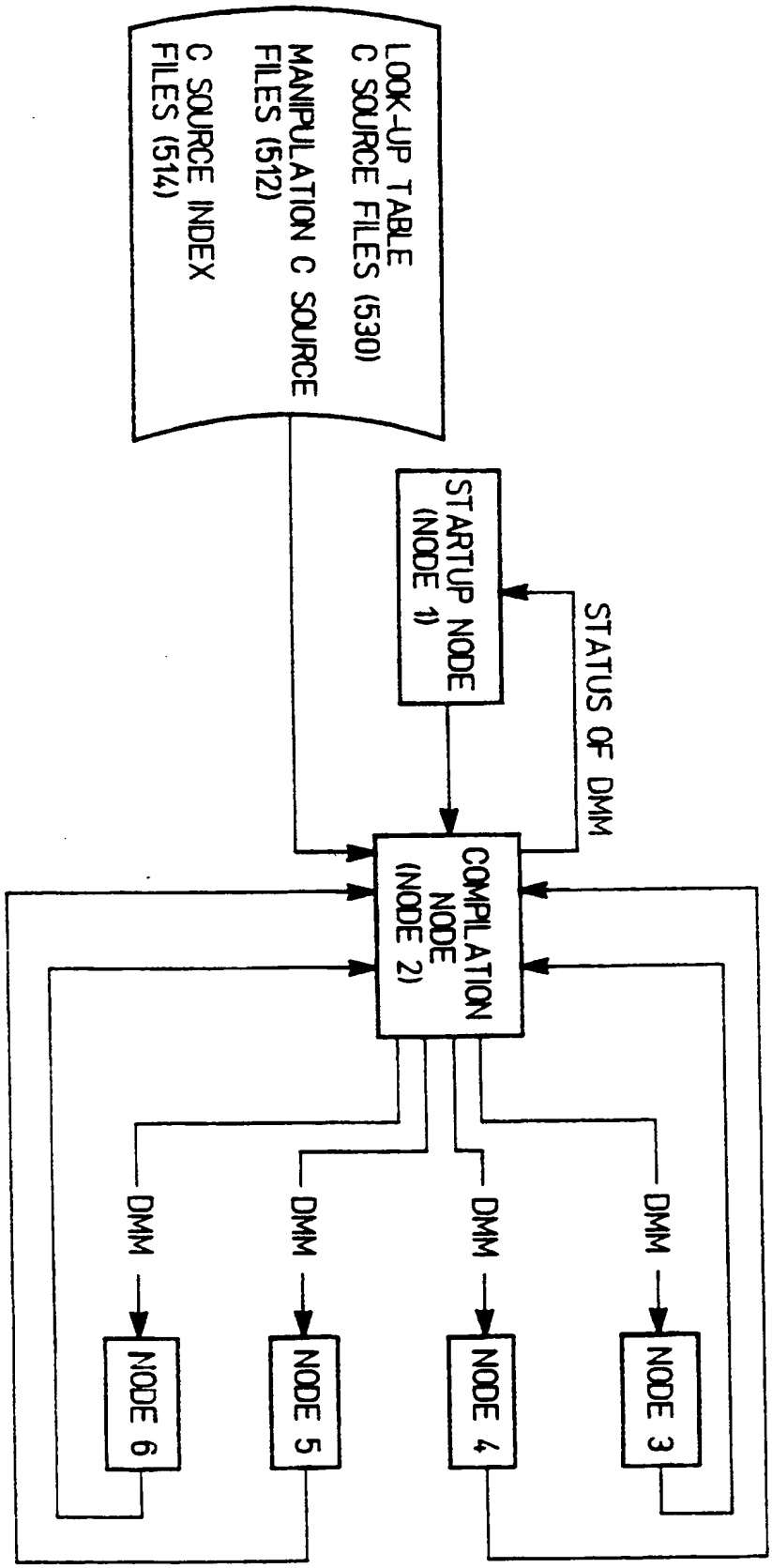


FIG 6

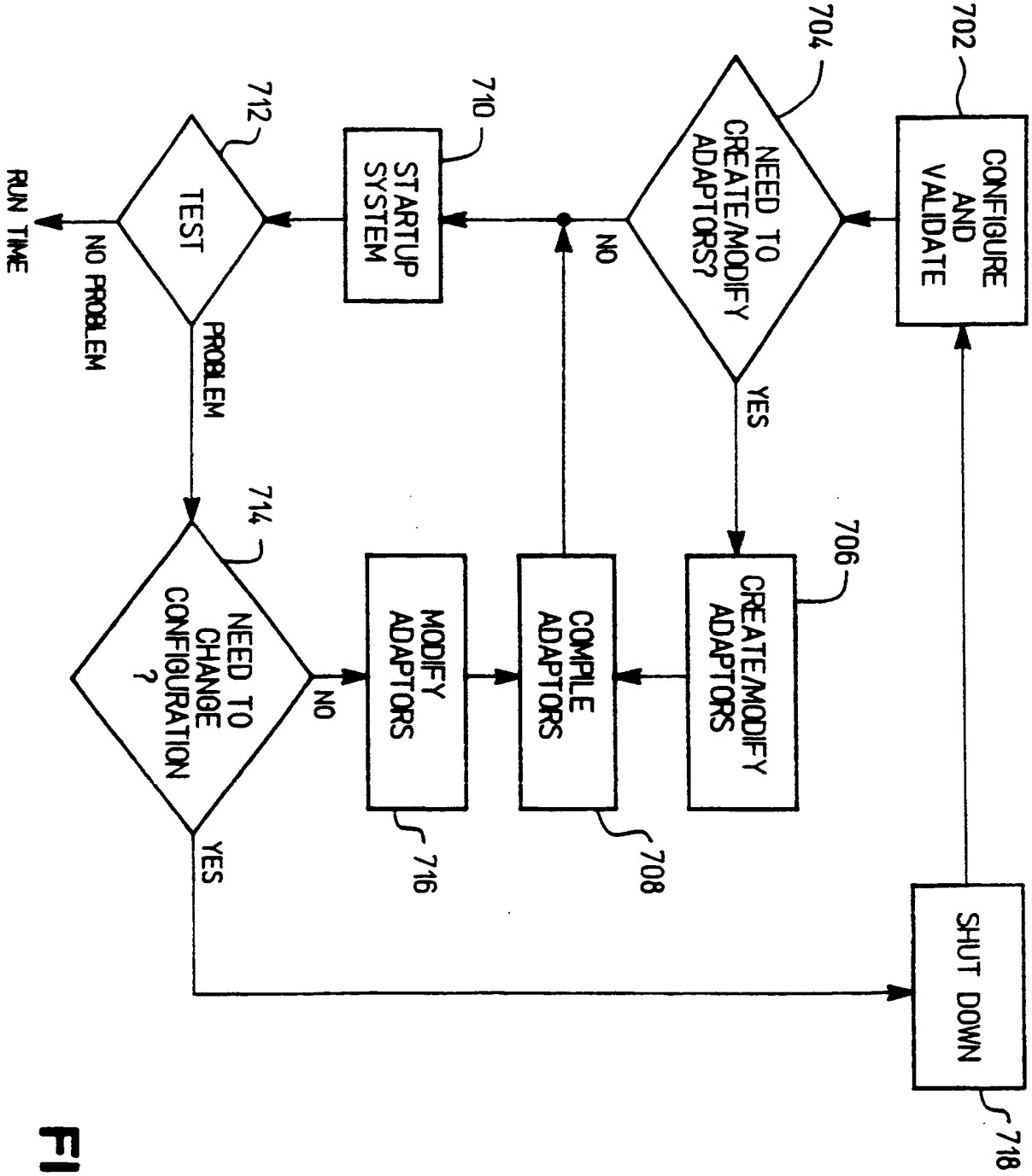


FIG 7

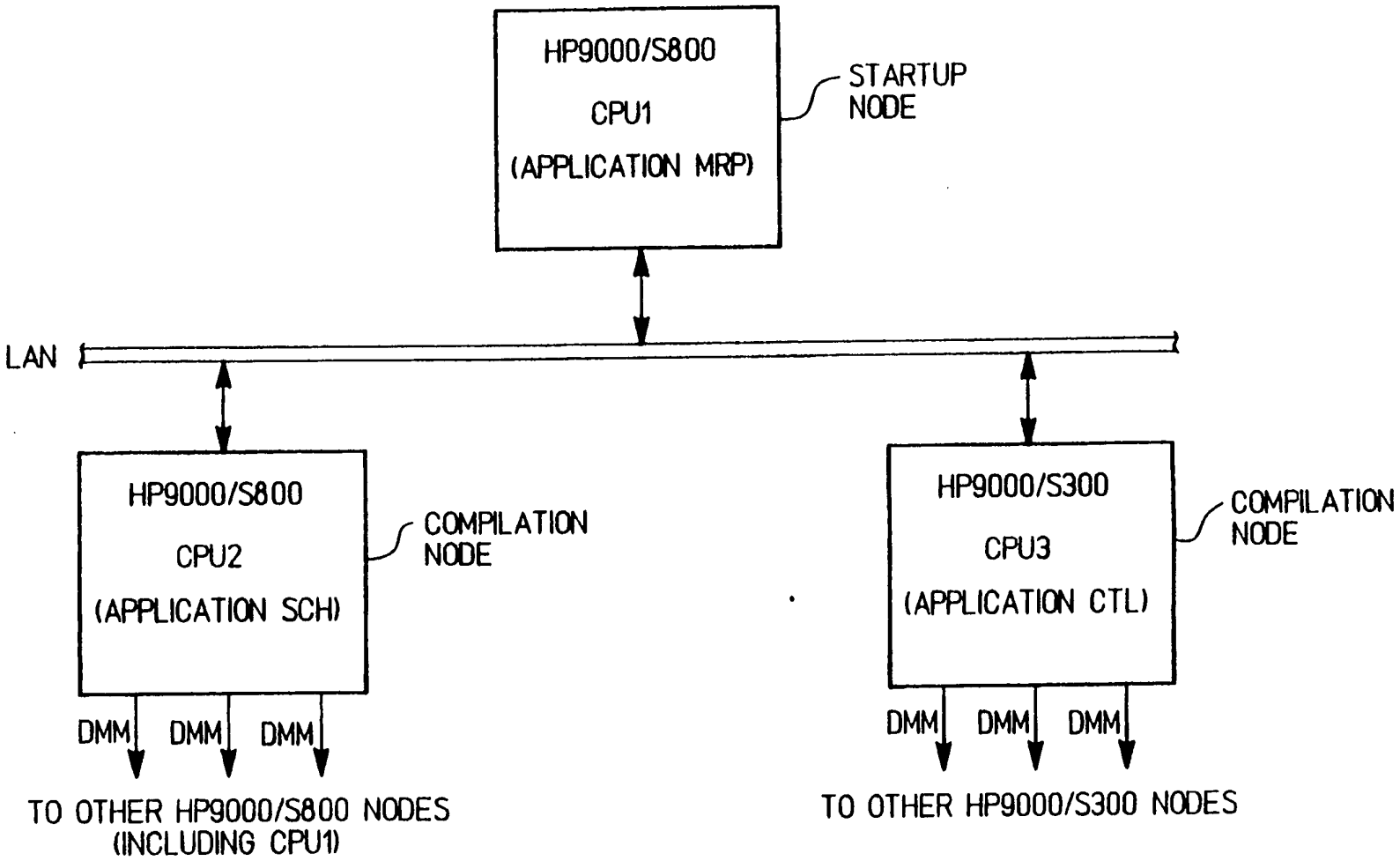


FIG 8

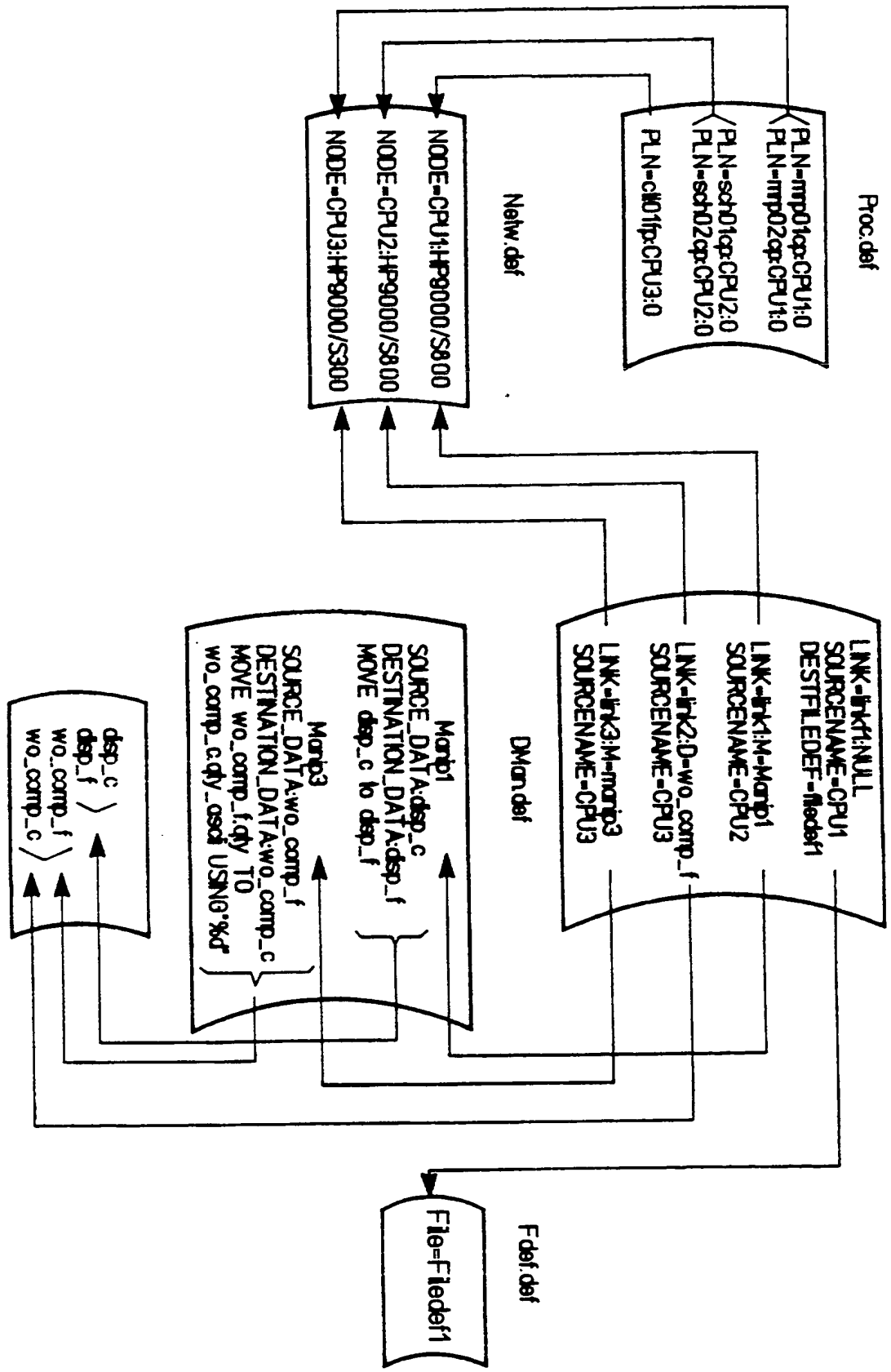


FIG 8

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 456 249 A3**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **91107604.0**

(51) Int. Cl.5: **G06F 9/46**

(22) Date of filing: **10.05.91**

(30) Priority: **10.05.90 US 521543**

(43) Date of publication of application:
13.11.91 Bulletin 91/46

(84) Designated Contracting States:
DE FR GB

(88) Date of deferred publication of the search report:
20.01.93 Bulletin 93/03

(71) Applicant: **Hewlett-Packard Company**
Mail Stop 20 B-O, 3000 Hanover Street
Palo Alto, California 94304(US)

(72) Inventor: **Pham, Thong**
10148 Judy Avenue
Cupertino, California 95014(US)
Inventor: **Gulland, Scott**
3681 Irlanda Way
San Jose, California 95125(US)
Inventor: **Amino, Mitch**
373 Bundy Avenue
San Jose, California 95117(US)

(74) Representative: **Baillie, Iain Cameron et al**
c/o Ladas & Parry, Altheimer Eck 2
W-8000 München 2(DE)

(54) System for integrating application programs in a heterogeneous network enviroment.

(57) A system which integrates applications that run on a plurality of homogenous or heterogeneous computers on a network. System Configuration files (510) in source code are created from a high level definition of the distributed system (LAN) which is to be integrated. The configuration files (510) include data such as the types and formats of data for each process (402) on each node (400) of the system, identification of all applications and machine types, topography and the data manipulations needed for sending messages and files and the like from an application program in a first computer language and of a first data type to an application program in a second computer language and of a second data type. Node-specific data manipulation modules (DMM 528) are formed at each node (400) during startup of the system, and these modules are automatically distributed to nodes (400) on the network having the same architecture. The invention allows applications having different physical data characteristics to communicate by using the data manipulation modules (DMM 528) so formed to manipulate the data at the source program into a common data representation (CDR) having data types common to all of the languages represented by the system and then reconverting the data to the local representation at the destination node.

EP 0 456 249 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number

EP 91 10 7604

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.5)
X	EP-A-0 304 071 (WANG LABORATORIES INC.) * page 47, line 47 - page 52, line 15 * * figure 9 *	1-4,9,10	G06F9/46
A	IDEM	5-8	
D,A	UNIX REVIEW June 1987, pages 66 - 75 H. JOHNSON 'EACH PIECE IN ITS PLACE' * the whole document *	1-10	
A	COMPUTER COMMUNICATIONS. vol. 13, no. 1, January 1990, LONDON, GB pages 4 - 16 H. LORIN 'APPLICATION DEVELOPMENT, SOFTWARE ENGINEERING AND DISTRIBUTED PROCESSING' * page 10, right column, line 53 - page 11, right column, line 18 * * page 12, right column, line 41 - page 15, left column, line 52 *	1-10	
			TECHNICAL FIELDS SEARCHED (Int. Cl.5)
			G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	19 NOVEMBER 1992	JONASSON J.T.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P0401)